ABSTRACT

Title of dissertation:	Tuning Parallel Applications in Parallel
	Ananta N Tiwari, Doctor of Philosophy, 2011
Dissertation directed by:	Professor Jeffrey K Hollingsworth Department of Computer Science

Auto-tuning has recently received significant attention from the High Performance Computing community. Most auto-tuning approaches are specialized to work either on specific domains such as dense linear algebra and stencil computations, or only at certain stages of program execution such as compile time and runtime. Real scientific applications, however, demand a cohesive environment that can efficiently provide auto-tuning solutions at all stages of application development and deployment. Towards that end, we describe a unified end-to-end approach to auto-tuning scientific applications. Our system, Active Harmony, takes a search-based collaborative approach to auto-tuning. Application programmers, library writers and compilers collaborate to describe and export a set of performance related tunable parameters to the Active Harmony system. These parameters define a tuning search-space. The auto-tuner monitors the program performance and suggests adaptation decisions. The decisions are made by a central controller using a parallel search algorithm. The algorithm leverages parallel architectures to search across a set of optimization parameter values. Different nodes of a parallel system evaluate different configurations at each timestep.

Active Harmony supports runtime adaptive code-generation and tuning for parameters that require new code (e.g. unroll factors). Effectively, we merge traditional feedback directed optimization and just-in-time compilation. This feature also enables application developers to write applications once and have the auto-tuner adjust the application behavior automatically when run on new systems. We evaluated our system on multiple large-scale parallel applications and showed that our system can improve the execution time by up to 46% compared to the original version of the program.

Finally, we believe that the success of any auto-tuning research depends on how effectively application developers, domain-experts and auto-tuners communicate and work together. To that end, we have developed and released a simple and extensible language that standardizes the parameter space representation. Using this language, developers and researchers can collaborate to export tunable parameters to the tuning frameworks. Relationships (e.g. ordering, dependencies, constraints, ranking) between tunable parameters and search-hints can also be expressed.

Tuning Parallel Applications in Parallel

by

Ananta N Tiwari

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2011

For my dad and mom, Ashok Nath Tiwari and Kamala Tiwari.
For the better half, Prathu.
For my sisters, Sunu and Sudhu.
For my nephews, Rohan and Raunak.

Acknowledgments

First and foremost, I would like to extend my sincere gratitude to my advisor, Dr. Jeffrey K. Hollingsworth. Without your guidance, patience and undivided support, this research would not have materialized. Thank you for your continued kindness.

I would also like to thank my friends, colleagues and my seniors (in no particular order) — Arkady Yerukhimovich, Dov Gordon, Vladimir Kolovski, Nick Rutar, Tuğrul İnce, Mike Lam, Ray Chen, Mustafa Tikir, Chadd Williams, Jeff Odom, Geoffrey Stoker, Vahid Tabatabaee and I-Hsin Chung. Thank you for always being there when I needed support. Thank you to Arkady, Dov and Vladimir for three wonderful years as housemates and for your perpetual friendship. Thank you to Nick for always cheering me up with his witty remarks and pop-culture references. Thank you to Tuğrul for always showing me the bright side of different situations. Thank you to Mike, Geoff and Ray for many interesting conversations. Let me tell you all that I will always cherish the friendship that we have developed for the rest of my life.

Sincere thank you to my family. Dad and mom, you have always been an infinite source of inspiration. Thank you to my dear wife, Prathana, who has always been there for me, who endured many all-nighters, and who always guided me in trying times.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Today's complex and diverse architectural features require applying nontrivial optimization strategies on scientific codes to achieve high performance. As a result, programmers usually have to spend countless hours in modifying and tuning their codes. Furthermore, a code that performs well on one platform often faces performance problems on another; therefore, the tuning process must be largely repeated to move from one computing platform to another. Recently, there has been a growing interest in developing empirical *auto-tuning* software that helps programmers manage this tedious process of tuning and porting their codes. Empirical auto-tuning software (or auto-tuners) can be broadly grouped into three categories: (1) compiler-based auto-tuners that automatically generate and search a set of alternative implementations of a computation [16, 95, 33]; (2) application-level auto-tuners that automate empirical search across a set of parameter values proposed by the application programmer [18, 62]; and, (3) runtime auto-tuners that automate on-the-fly adaptation of application-level and architecture-specific parameters to react to the changing conditions of the system that executes the application [11, 87]. What is common across all these different categories of auto-tuners is the need to *search* a range of possible configurations to identify the best-performing solution. The resulting search space of alternative configurations can be very complex and prohibitively large. Therefore, a key challenge for auto-tuners, especially as we expand the scope of their capabilities, involves scalable search among alternative configurations.

While it is important to keep advancing the state-of-the-art in auto-tuning software from the above three categories, our research demonstrates that full applications require a mix of these rather disjoint tuning approaches: compiler-generated code, application-level and runtime parameters exposed to auto-tuning environments. This brings us to the thesis of our research: *Full applications demand and benefit from a cohesive environment that can seamlessly select between different kinds of auto-tuning techniques and that employs a scalable search to manage the cost of the search process.*

To investigate the thesis of our research, we have introduced several extensions to a scalable end-to-end auto-tuning infrastructure — *Parallel Active Harmony* (henceforth referenced as Active Harmony). Active Harmony takes a search-based collaborative approach to auto-tuning. Application programmers, library writers and compilers collaborate to describe and export a set of performance related tunable parameters to the Active Harmony system. These parameters define a tuning search-space. The auto-tuner monitors the program performance and suggests adaptation decisions. The decisions are made by a central controller using a parallel search algorithm. The algorithm leverages parallel architectures to search across a set of optimization parameter values. Different nodes of a parallel system evaluate different configurations at each timestep.

Active Harmony can be used to tune compiler-level parameters, application-level input parameters and runtime parameters. Furthermore, our system supports runtime adaptive code-generation and tuning for parameters that require new code (e.g. loop unroll factors, loop permutation orders). *Effectively, we merge traditional feedback directed optimization and just-in-time compilation.* This feature also enables application developers to write applications once and have the auto-tuner adjust the application behavior automatically when run on new systems.

In the next section, we motivate the need for a search-based auto-tuner. We then identify various sources of tunability in real codes and discuss how those sources can be exploited by auto-tuners to find optimal parameter configurations faster.

1.1 Motivation

To motivate the need for auto-tuning, we consider the problem of tuning matrix multiplication, a well-known and well-studied benchmark kernel. We apply two code-transformations to this kernel — tiling[1] and unrolling[2]. More precisely, we take a naïve IJK-implementation and tile all three loops with the same tile-size and unroll the innermost loop. Tiling factors range from 2 to 80 and unrolling factors from 2 to 32. Figure 1.1 illustrates how tiling and unrolling transformations interact with each other. We elaborate on these interactions in chapter 6. The figure shows the performance of square matrix multiplication (of size 800×800) as a function of tiling and unrolling factors. The search space is not smooth and contains multiple

[1]Tiling loop transformation partitions a loop nest's iteration space into small blocks and then iterates through those blocks in sequence. The goal is to maintain a small data footprint for the sub-loop nests for better cache usage.

[2]Unrolling replicates a loop body by a given factor and then steps the loop by the same factor. Unrolling simplifies the control flow and exposes independent computations to the scheduler to improve instruction-level parallelism.

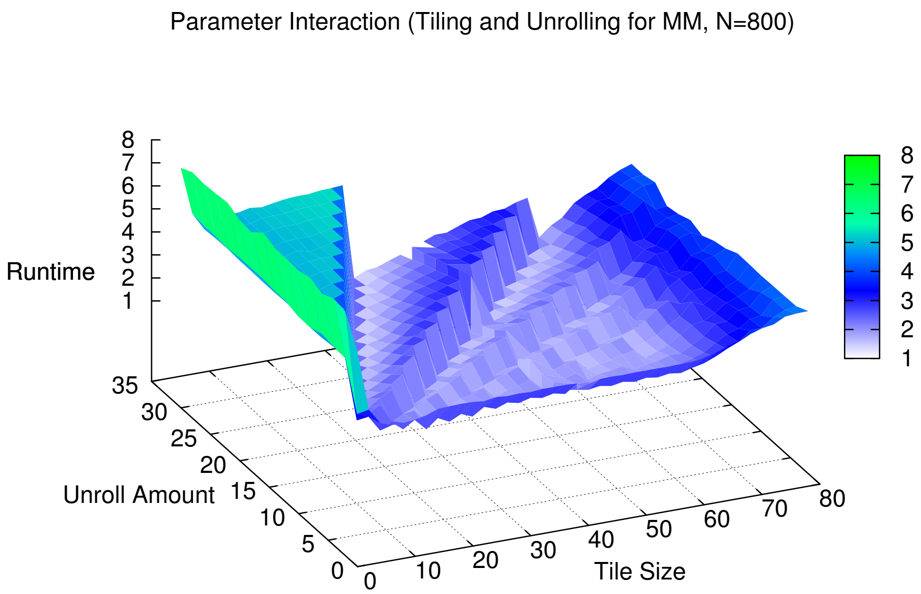


Figure 1.1: Parameter space for tiling and unrolling for MM

minimums and maximums. The best and the worst configurations are a factor of six different.

This example demonstrates the need for search-based auto-tuning systems to adapt code. The relatively large plateau of good performance is also typical of many applications we have studied. Such a topography argues for the need to have auto-tuning systems that rapidly get applications out of "bad" regions of performance and into the "good" ones. Achieving optimal performance is a secondary goal once "good" performing region is reached.

1.2 Auto-tuning Modes

Auto-tuning can be performed in either offline or online mode. Offline mode refers to tuning between successive full application runs. The performance data collected in this fashion can be used for training based application tuning as well. This mode is often used for parameters that are read once when the program starts and must remain fixed throughout the execution of the application. For example, a parameter which governs data distribution can be tuned in an offline mode to minimize communication costs. In addition, the offline mode can also be used to for auto-tuning compiler-generated code.

Online mode refers to runtime tuning during production runs. This mode is used when application parameters can be adapted during runtime for improved performance. For example, in a 3D Jacobi algorithm, blocking factors for the triply nested stencil loops can be changed across different iterations to get a better cache reuse. Online tuning has the privilege of exploiting fine-grained, up-to-date and accurate performance data that can be directly linked back to specific code sections, characteristics of input datasets, architecture specific features and changing conditions of the system.

1.3 Sources of Tunable Data

Auto-tuners do not randomly create things to change in a program. Rather they provide a way optimize a program's performance based on trying out a set of possible changes. This set of possible changes comes from many sources including options that application programmers discover, library parameters and algorithmic choices that depend on library use, and from possible code transformations identified by compilers.

In addition to considering what can change, another important source of outside guidance for auto-tuning is insight on how changes might impact performance. Information from performance models that are parametrized by the tunable components can help guide an auto-tuning system to the correct values to use. Even simple non-parametric models can help to indicate how close an auto-tuning system is to reaching the best achievable performance. We provide a simple language that programmers, library writers, compilers and performance modeling experts can utilize to express this information to the auto-tuners. The language is presented in chapter 4.

Another source of data for tuning is to use training runs to auto-tune the software based on a well defined benchmark or typical workload. Training runs are application executions designed mainly to produce performance data that feed into the auto-tuning process. A single auto-tuning run can evaluate several configurations. More often than not, several auto-tuning runs are performed with slightly different settings (e.g. different input sizes) before the application is run under production mode. Therefore, if a database of such evaluations (along with some context information) is maintained locally, tuning time can be reduced by consulting the database during consecutive auto-tuning runs. We use this technique in Active Harmony. A more elaborate discussion is provided in chapter 4.

Target architecture parameters such as cache sizes, memory bandwidth, register file size etc. can be used rule out mediocre configurations from the parameter space. For example, the tile-size parameter for tiling loop transformation can be constrained by half of the available cache. In addition, runtime tuning can take advantage of the input dataset knowledge to further reduce the search space. An example of this would be to opt out from data-copy optimization for matrix multiplication if the input matrices fit in the largest available cache. We elaborate more on how this architecture-specific and runtime information is used by Active Harmony in chapters 6, 7 and 8.

In this dissertation, we make the following contributions:

1. We present a unified end-to-end tuning solution for scientific applications and show that full applications benefit from such a system. Our system can be used to tune compiler-level parameters, application-level input parameters and runtime parameters.

2. We present a powerful parameter tuning algorithm that leverages parallelism to converge to solutions faster. Multiple, sometimes unrelated, configurations are evaluated simultaneously at each search step.

3. We study performance variations inherent in today's HPC architectures and suggest robust strategies that function properly, even when application performance is variable.

4. We present an auto-tuning approach that supports runtime adaptive code-generation and tuning for parameters that require new code. This feature enables programmers to write applications once and have the auto-tuner adjust the application behavior automatically when run on new systems or on the same system with a new workload.

5. We demonstrate how our system allows simultaneous tuning of multiple application components.

6. We present a simple interface for application programmers, library writers, compilers and performance modelers to interact with auto-tuners. This part of our work is aimed at bridging the communication gap that exists between these communities.

Chapter 2

Related Work

One of the main contributions of this work is the integration of three disjoint auto-tuning techniques — application-level input parameter tuning, compiler-based tuning, and runtime tuning. The integrated framework is general-purpose and can be used to tune whole applications. We divide the projects that are related to our work into seven broad categories — domain-specific auto-tuners, compiler-based auto-tuners, application-level auto-tuners, runtime auto-tuners, auto-tuning via performance modeling, and search algorithms for high-dimensional optimization spaces. Finally, we discuss relevant projects that have studied performance variability in high performance computing platforms.

2.1 Domain-specific Auto-tuners

There are many research projects working on empirical optimization of linear algebra kernels and domain specific libraries. ATLAS [89] uses the technique to generate highly optimized BLAS routines. The Oski (Optimized Sparse Kernel Interface) [88] library provides automatically tuned computational kernels for sparse matrices. FFTW [30] and SPIRAL [93] are domain specific libraries. FFTW combines the static models with empirical search to optimize FFTs. SPIRAL generates empirically tuned Digital Signal Processing (DSP) libraries.

Datta et al [24] explore architecture-specific optimizations for stencil computations — a class of algorithms that are at the heart of many structured grid codes. They use a Perl code generator that produces multithreaded C code variants which utilize their stencil computation optimization strategies. In addition, they use a search-based component that navigates the parameter space through a combination of explicit search for the global maximum with heuristics that constrain and prune the search space.

Williams et al [90] apply the idea of search-based auto-tuning to lattice Boltzmann application, which is characterized by complex data structures and memory access patterns. The authors use a code generator that generates variants. These variants use various optimization strategies such as loop unrolling and reordering, simdization, software prefetching, TLB blocking etc. The system then identifies highly optimized platform-specific versions for a variety of architectures.

Chandramowlishwaran et al [12] look into single-node performance optimization, tuning and analysis of the fast multipole method (FMM) on a diverse set of multi-core systems. They consider numerous performance enhancing strategies for FMM — low-level instruction selection, SIMD vectorization and scheduling, numerical approximation, data structure transformations, OpenMP-based parallelization, and tuning of algorithmic parameters.

Domain-specific libraries, in most cases, are better suited than Active Harmony to handle domain-specific computations. Our goal is different in that we provide a general-purpose compiler based framework, which can generate and evaluate different optimizations that can be applied on arbitrary application codes.

2.2 Compiler-based Auto-tuners

Compiler-based auto-tuning frameworks mostly target loop optimizations that can exploit available memory hierarchy. Tuning is done by generating a set of equivalent alternatives and by searching/selecting the one that performs best on the target architecture. Apart from an efficient search algorithm to navigate the search space of loop transformation parameters (e.g. tiling factors, loop permutations), another key requirement for a compiler-based auto-tuning framework is a code-generation framework that can generate code rapidly during the search. In this section, we first discuss compiler frameworks that encourage application developer and expert participation in deciding what optimization strategy to use for a given piece of code. We then discuss projects that use these frameworks to do empirical tuning.

2.2.1 Code-transformation Frameworks

POET [94] is a transformation scripting language embedded in an arbitrary programming language. It is interpreted by a POET compiler to apply source-to-source code transformations. TLOG [45] is a code generator for parameterized tiled loops where tile sizes are symbolic parameters. Symbolic tile-size enables static or runtime tile size optimization without repeatedly generating the code and recompiling it for each tile size. Interactive Compilation Interface (ICI) [31] provides a flexible and portable interface to internal compiler optimizations so that iterative optimization [3] can be applied at the loop or instruction-level by adjusting optimization decisions externally.

Several tools for loop transformations and code generation are based on polyhedral representation of loops. Such representation facilitate compilers to compose complex loop transformations in a mathematically rigorous way to ensure code correctness. WRaP-IT [33] and Petit [43] are both polyhedral loop transformation frameworks that support composition of transformations. LeTSeE [71] is an iteration optimization tool based on the polyhedral model. It finds all legal affine scheduling of a loop nest and explores this space to find the best scheduling and parameter values. Pluto [9] is an automatic parallelization and locality optimization tool also based on the polyhedral model.

CHiLL [14, 15] is another polyhedral loop transformation and code generation framework. CHiLL provides capability for composing high-level loop transformations with a simple script interface to describe the transformations and search space to the search engine. It supports a wide array of loop transformations (for both perfect and imperfect loop nests) required to achieve high-performance on today's computer architectures. We use CHiLL in our compiler-based auto-tuning work.

2.2.2 Auto-tuning Frameworks

In this section, we discuss relevant projects that use search algorithms to explore compiler generated parameter spaces. Orio [35] is an extensible annotation system, implemented in Python, that aims to improve both performance and productivity by enabling software developers to insert annotations into their source code (in C or Fortran) that trigger a number of low-level performance optimizations on a specified code fragment. The tool generates many tuned versions of the same operation using different optimization parameters, and performs an empirical search for selecting the best among multiple optimized code variants.

Kisuki et al [46] address the problem of selecting tile sizes and unroll factors simultaneously. Different search algorithms are used to search the parameter space — Genetic algorithms, Simulated Annealing, Pyramid search, Window search and Random search. Qasem et al [74, 72] use a modified version of pattern-based direct search algorithm to explore the same search space. In addition, they provide compiler-based tuning for whole applications. The framework performs an offline search on compiler generated search spaces.

Triantafyllis et al [84] use iterative compilation technique to explore the space of optimizations offered by compilers. The framework uses the compiler writers' knowledge to select a small number of promising optimization alternatives for a given code segment. Time required for multiple compilations is kept under control by evaluating only these alternatives for hot code segments.

Our work on compiler-based auto-tuning considers a much broader range of loop transformations and our auto-tuner can also effectively navigate the space of optimizations offered by compilers. The design of our system separates the search algorithm from the code-generation part, which allows us to easily switch between different underlying code-generation tools (e.g. if we are tuning CUDA code, we can switch to a code-transformation framework that supports CUDA code transformation). Finally, Kisuki et al[46] report converging to a solution in hundreds of iterations. By effectively utilizing the underlying parallel infrastructure, we converge to solutions in a few tens of iterations.

2.3 Application-level Auto-tuners

Scientific applications and libraries expose a set of input parameters which allow the end-users to adjust their execution behavior to the characteristics of the execution environment. The selection of appropriate parameter values is, thus, key in ensuring maximum throughput. Application-level auto-tuning frameworks attempt to identify a set of input parameters that delivers a reasonable performance.

Nelson et al [62] focus on empirical techniques to find the best integer value of application-level input parameters that the programmer has identified as being critical to performance. Programmers provide high-level models of the impact of parameter values, which are then used by the tuning system to guide and to prune the search for optimal input parameters. Hunold et al [38] consider input parameter tuning for the matrix-matrix multiplication routine PDGEMM of ScaLAPACK [17]. PDGEMM is a part of PBLAS, which is the parallel implementation of BLAS (Basic Linear Algebra Subprograms) for distributed memory systems. Input parameters selected for experimentation include the dimension of input matrices, number of processors, logical processor grid and three blocking factors (one for each dimension). Experimental evaluation provides results with up to 47% increase in performance for PDGEMM.

Chung et al [19] use short benchmarking runs to reduce the cost of full application execution incurred in application-level input parameter tuning. Short benchmarking runs exercise all aspects of the application (or at least all aspects that are influenced by the choice of input parameters). The authors use their search algorithm (modified Nelder-Mead search) to navigate input parameter space of large-scale scientific applications. Similar ideas of using program reduction methods to tune application-level input parameters have also been reported by Che et al [13].

In our application-level input parameter tuning and offline tuning of whole applications, we utilize the idea of the short benchmarking runs. The main difference between the related projects and our system is that we use a parallel search algorithm to find better performing parameter configurations at a much faster rate.

2.4 Runtime Auto-tuners

Several techniques have been put forth to dynamically adapt a program to a given input and the runtime environment. Autopilot [75] is an online tuning framework for parallel applications. Based on application request patterns and observed system performance, Autopilot's real-time adaptive control mechanism automatically chooses and configures resource management algorithms. There are three main software components — "sensors", "decision-procedures" and "actuators". Sensors are used to extract quantitative and qualitative application performance data. Decisions are made by the decision-procedures using fuzzy logic, where are then executed by actuators by changing parameter values of applications and resource management policies of the underlying system.

AppLes [8] and Odyssey [65] both are application-centric tools and emphasize application level resource awareness. Applications adapt by (re)allocating the resources based upon a customized scheduling to maximize their performance. Rather than leaving the adaptation decisions to applications, Active Harmony's approach uses a centralized server to control such decisions and to coordinate the tuning of multiple application components.

CUMULVS [32] is an infrastructure library that allows a programmer to easily extract data from a running parallel simulation and send the data to a visualization package. CUMULVS includes the capability to steer user-defined parameters in a distributed simulation. The programmer defines the decomposition of parallel data and the number and types of scalar parameters, and CUMULVS does the rest. It supports the simultaneous viewing of multiple dynamically attached front-end visualization programs, or viewers.

SciRUN [67] is a scientific programming and debugging environment that allows interactive steering of large scale applications. SciRUN enables scientists to design and modify models and automatically change parameters and boundary conditions as well as the mesh discretization level needed for an accurate numerical solution. Our approach in Active Harmony is different because we do not attempt to alter the execution in a manner that can change the output of the program. We only consider performance-related parameters that do not affect the program's output or correctness.

Dynamic code-generation and runtime loading of different versions of code-sections is a technique that has been used both in the context of dynamic software updating [60] and auto-tuning [87]. ADAPT [87] is a compiler-supported infrastructure for high-level adaptive program optimization. It allows developers to leverage existing compilers and optimization tools by describing a runtime heuristic for applying techniques in a domain specific language, ADAPT Language. ADAPT supports remote dynamic compilation, parameterization and runtime sampling, allowing developers the flexibility in heuristic development. Our work is distinct from ADAPT in two ways. First, we target SPMD-based parallel applications and use the parallelism to our advantage by evaluating multiple parameter configurations within one search iteration. Second, our system is designed to tune multiple code-sections simultaneously.

CPO (Continuous Program Optimization) [11] uses monitoring agents that collect information from all layers of an application execution stack. This information is used to model the application behavior and predict the relative benefit of using large pages for different application data structures.

MATE [57] performs dynamic tuning in three basic and continuous phases: monitoring, performance analysis and application modification. This environment dynamically and automatically instruments a running application to gather information about the application's behavior. The analysis phase receives events, searches for bottlenecks, detects their causes and gives solutions on how to overcome them. Finally, the application is dynamically tuned by applying a solution described by simple performance models. Our work is distinct from MATE in that we use effective and light-weight algorithms to search for optimal parameters rather than relying on simple performance models.

PetaBricks [4] is an implicitly parallel language and compiler that allows a programmer to describe algorithmic choices explicitly; the idea is to allow the compiler to perform deeper optimization. Using this mechanism, PetaBricks programs define a search space of possible algorithmic program pathways. At runtime, an auto-tuner composes the program pathways using fine-grained algorithmic choices and finds the right choice for many other parameters.

Otto et al [66] present a language-based auto-tuner that targets applications that use task and pipeline parallelism. Their approach automatically infers tuning parameters from high-level parallel language constructs. Instead of identifying and adjusting tuning parameters manually, the system exploits compiler's context knowledge about the program's parallel structure and appropriate heuristics to figure out the values for tunable parameters at runtime. We have designed the Active Harmony API so that the changes that must be introduced in an application to make it tunable are minimal. Language-based auto-tuners, in most cases, require application developers to make extensive code modifications.

2.5 Auto-tuning via Performance Modeling and Analysis

Many groups [39, 52, 44, 79, 81], have used modeling techniques to study application behavior and to predict application performance. Ipek et al [39] uses machine learning techniques to build performance models. The idea is to build the model once and query it in the future to derive performance prediction for different input configurations. The model is trained on a dataset which consists of points spread regularly across the complete input parameter space. After the training phase, they query their model at runtime for points in the full parameter space. The paper reports that their model can predict application performance for points not included in the training dataset within 5-7% of actual application runtime. Others [79] and [52] have also used similar machine learning techniques to build performance models. The key limiting factor of the listed modeling techniques is the size of the training set. Ipek et al report using training set of size $3K$ (for SMG2000), which amounts to $3K$ unique executions of the application just to train the model. Active Harmony avoids the cost of these training runs and can improve application performance within one single execution.

Kerbyson et al [44] present predictive analytical model that accurately reflects the behavior of SAGE, a multidimensional hydrodynamics code with adaptive mesh refinement. Inputs to the model include machine-specific information such as latency and bandwidth and application-specific information such as data-decomposition, problem size etc. Vetter et al [86] use the idea of performance assertions. Performance assertions are expressions that contain a variety of tokens that represent empirically measured performance metrics, constants, variables, mathematical operations etc. By allowing the user to specify a performance expectation with individual code segments, the runtime system can jettison raw data for measurements that pass their expectation, while reacting to failures with a variety of responses. Static models, performance expectations and assertions become outdated when architectures and compilers change. Moreover, since models usually make a lot of simplifying assumptions, they can be rendered ineffective as the size and the complexity of the problem being solved changes.

There are several tools that provide performance analysis of parallel programs. Scalea [85] is a performance instrumentation, measurement, analysis and visualization tool for parallel programs that supports post-mortem and online performance analysis. A graphical user interface provides an access to performance metrics at the level of arbitrary code regions, threads, processes, and computational nodes. KappaPI 2 [42] tool is a performance analysis tool designed to incorporate parallel performance knowledge about performance bottlenecks. The tool is able to detect and analyze performance bottlenecks and then make suggestions to the user to improve the application behavior.

TAU [78] is a portable profiling and tracing toolkit for parallel programs. TAU is capable of gathering performance information through instrumentation of functions, basic blocks and statements. A graphical interface allows users to explore the performance data and to pinpoint potential performance problems. HPCToolkit [2] is an integrated suite of tools that supports measurement, analysis, attribution and presentation of application performance for both sequential and parallel programs. The tool can identify and quantify scalability bottlenecks in fully optimized parallel programs. Active Harmony utilizes the information presented by these performance analysis tools to determine whether there are opportunities for performance improvement.

2.6 Search Algorithms

In this section, we briefly review some search algorithms that have been used in various auto-tuning frameworks. While this is not a complete set of search algorithms used in the auto-tuning realm, we describe the most widely used algorithms. ATLAS [89] uses orthogonal line search, which optimizes each tunable parameter independently by keeping the rest fixed to their reference values. The parameters are tuned in a pre-determined order and each successive parameter tuning uses the optimized values for parameters that precede it in the ordering. The disadvantage of using this search in a general-purpose framework is that it requires a pre-determined ordering for parameters. ATLAS exploits years of experience in dense linear algebra tuning to determine appropriate ordering for parameters. However, such knowledge is not available for general-purpose tuning cases.

Several auto-tuners [46, 20, 50] have used genetic algorithms (GA). GA algorithm starts by randomly generating an initial population of possible configurations. Each configuration is represented as a genome and the "fitness" of the configuration is the performance metric. Based on the fitness, each successive iteration of the algorithm produces new set of configurations by using genetic operations — mutation, crossover and selection. While GA has shown its promise by converging to good configurations, the key disadvantage lies in its long convergence time. Furthermore, the transient behavior of GA is unpredictable and jittery, which makes the algorithm unsuitable for online tuning of production codes.

Direct search methods are also popular among auto-tuners. These methods do not explicitly use function derivatives. The parameter tuning problem is a very good use-case for direct search methods, since in most cases the performance function at a given point in the search space have to be evaluated by actually running the program. The Nelder-Mead Simplex algorithm [61], is one of the most widely used direct search methods in auto-tuning systems [1, 23, 35]. We provide a high-level description of the algorithm in chapter 3. The Nelder-Mead algorithm sometimes finds solutions efficiently, however, many studies [51, 56, 47] have described unpredictable behavior of the algorithm as the number of parameters (search space dimension) increases.

Looptool [74], which is a compiler-based auto-tuning framework uses pattern-based direct search method proposed by Hookes and Jeeves [37]. The pattern-based search method have been observed to be very reliable method, however, in some cases, the convergence time of the algorithm is slow [47].

2.7 Performance Variability

Performance variation in parallel architectures is discussed by Kramer et al[49]. The authors explore the amount of variation seen in large distributed memory systems. They analyze the causes for the observed variations and discuss what can be done to decrease the variations without impacting performance. Petrini et al [69] attempt to quantify performance variability. The authors use models to quantify the gap between measured and expected application performance. Their approach also evaluates the contribution of various sources of variability to the overall application behavior.

Mraz et al [59] look into the performance variability in point-to-point communication. This work pinpoints several causes for this variability — daemons, interrupts, and other system activity. They also analyze multiple techniques to reduce the performance loss (e.g. raising the priority of user application above that of the system daemons).

Most of the published work in studying and quantifying performance variability in real systems provide a systematic mechanism to find the causes of variations, and then discuss strategies to fix them. Our approach is different; we model performance variations as a stochastic process and suggest strategies that function properly, even when application performance is variable.

Chapter 3

Parameter Tuning Algorithm

A key to any auto-tuning system is how it goes about selecting the specific combinations of choices to try. We refer to this process as the search algorithm. While a simple parameter space might be exhaustively searched, most systems contain too many combinations to try them all. Instead, an auto-tuning system must rely on search heuristics to evaluate only a sub-set of the possible configurations while trying to find an optimal one (or at least as nearly optimal as practical). In this chapter, we focus on the design of this algorithm. Parameter tuning problems have some distinct characteristics and requirements that must be considered when designing the optimization algorithm. The characteristics and requirements are:

1. **Exponential size of the Optimization Space:** Consider an application with 10 tunable parameters. If each of the tunable parameter takes 4 values, the size of the parameter space is 2^{20}. Our optimization algorithm should be capable of tuning multiple parameters at one time while exploring only a small fraction of the search space.

2. **Unstructured Optimization Space:** Many of the optimization algorithms work appropriately on well structured problems. For instance, gradient based algorithms are appropriate for continuous convex optimizations. In Active Harmony, we are often dealing with integer restricted parameters and an un-

known non-convex function with multiple local minimums. Therefore, we have to resort to a class of optimization algorithms that work under these conditions.

3. **Parallel Search:** We are primarily interesting in tuning high performance scientific applications. These applications are run on parallel computing platforms. Therefore, it is desirable to employ optimization algorithms that can take advantage of the underlying parallelism by searching for configurations in parallel.

4. **Performance Variability:** The tuning process monitors the performance of the application for different tunable parameter values and based on the observed performance measurements modifies the parameters. The goal is to ultimately find the parameter values that gives optimal, or at least, near optimal performance. Most optimization algorithms assume perfect and accurate data, which is not the case in real systems. The performance measurement for an application with a fixed set of tunable parameters varies in time even on the same platform. These variations are triggered by external factors such as OS jitter. Therefore, it is desirable to develop algorithms that are resilient and converge to good solutions, even in the presence of performance variability.

5. **Performance Metric:** The final result and the convergence time (in terms of the number of evaluations of the objective function) are two common performance metrics for optimization algorithms (used by the optimization algorithms community). However, for auto-tuning systems, these are not the most

appropriate ones. For auto-tuning systems, the overall performance of the system from the start to the end are equally important. This is particularly critical for online optimization because if the transient behavior is poor, the overall tuning time and hence the running time of the application will suffer. Therefore, the appropriate performance metric should consider and capture the transient behavior and performance of the intermediate points visited in the path to the final solution.

6. **Parameter Interactions:** Tunable parameters interact with each other in complex ways. For example, for loop-level transformation parameters, tile-size parameter can limit unroll-factor parameter. Our parameter search algorithm must take into account such interactions and be able to tune the parameters simultaneously.

In the next section, we provide a problem definition for parameter tuning and introduce the performance metric for the optimization algorithms.

3.1 Problem Definition and Performance Metric

We consider software applications with an iterative structure using SPMD style computation. After finishing each iteration on all processors, information is exchanged between nodes and the next iteration starts. We consider that the application should run for a fixed number of iterations or timesteps, say K, to get the ultimate result. Our objective is to minimize $Total\_Time(K)$, which is the time that it takes to run the program for the desired number of timesteps, K. Suppose that we want to run the application for K timesteps on P parallel processors. Let T_k be the time that it takes to run the kth iteration, and $t_{p,k}$ be the time that it takes to run kth iteration on the pth processor. We have,

$$T_k = \max_{p=1,\cdots,P}(t_{p,k}), \tag{3.1}$$

and

$$Total\_Time(K) = \sum_{k=1}^{K} T_k. \tag{3.2}$$

Our main performance metric is $Total\_Time(K)$. Two typical performance metrics for optimization algorithms are the final solution after convergence and the convergence speed. These asymptotic metrics could be misleading for auto-tuning, since the number of iterations is fixed to K and the algorithm may not have enough time to converge. Even if K is large enough that the algorithm converges, the overall performance could be dominated by the performance of the algorithm at the initial timesteps before convergence.

Furthermore, for every iteration k, the worst case performance (maximum value) is used for T_k. This is again different from common practice, where the best performance is considered in every iteration. Note that in our auto-tuning model all processors should wait for the last processor before starting the next iteration; hence, the worst case performance is the bottle neck [1].

To clarify the performance metric issues, consider three optimization algo-

[1]Our actual tuning system works for applications that do not have this synchronization requirement, but for the purpose of algorithm analysis it provides the worst case scenario.

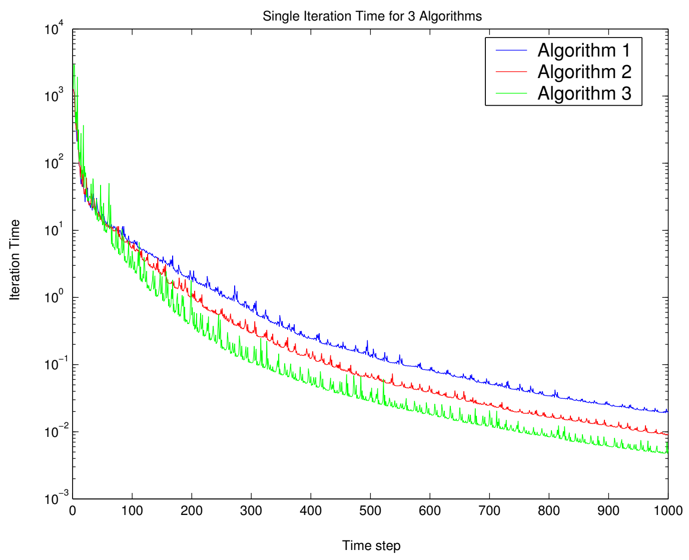


Figure 3.1: Single iteration time plot for 3 algorithms

rithms whose performance are plotted in figures 3.1 and 3.2. These algorithms are different variants of direct search optimization methods that we discuss later in this chapter. Figure 3.1 shows each iteration's worst case performance, T_k, versus iteration. This plot is closer to the plots that are typically used for comparison of the optimization algorithms, where the best case performance is usually used. However, for auto-tuning the Total Time plot of figure 3.2 is more appropriate. Note that the Total Time curve of figure 3.1 is the integral of the worst case performance given in figure 3.1. By looking at figure 3.1, one may conclude that Algorithm 3 is the best, where as from figure 3.2, we can conclude that Algorithm 1 is more appropriate for auto-tuning. The main reason behind the discrepancy is the transient behavior of Algorithm 1 in the first 100 timesteps, where it has significant fluctuations.

In summary, for auto-tuning, initial transient behavior of the algorithm can be more significant than the final value at the convergence point of the algorithm. This fact should be taken into account in selecting the appropriate optimization

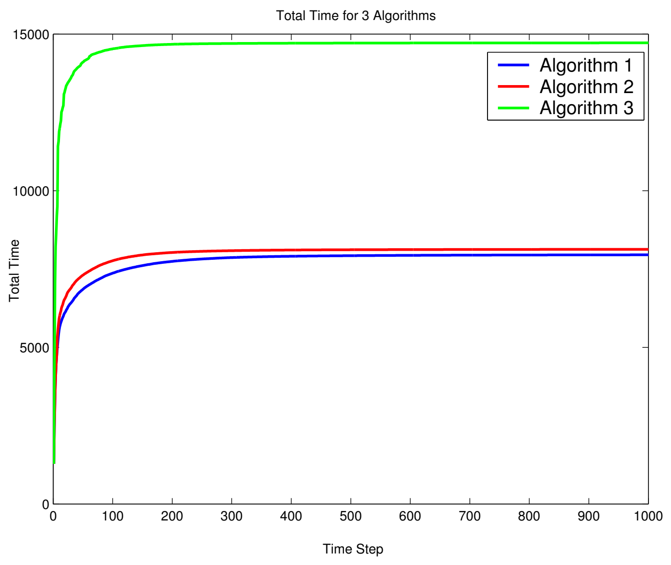


Figure 3.2: Total time plot for 3 algorithms

algorithm. For instance, randomized scheduling algorithms such as Simulated Annealing and Genetic Algorithms that are often considered suitable for unstructured optimization problems are not appropriate for auto-tuning, since even though they can ultimately converge to the optimal solution, they have very poor initial performance.

3.2 Direct Search Algorithms

Direct search methods are a class of optimization algorithms that do not explicitly use function derivatives. Consider the problem of finding a local minimum of a real-valued function $f(X)$. If f is differentiable and $\nabla f(X)$ can be computed or estimated, there is a plethora of gradient-based algorithms to solve this problem. However, if $\nabla f(X)$ is not available or if f is not differentiable, we have to rely on alternative algorithms such as direct search methods. The parameter tuning problem is a very good example for the latter group, since in most cases the performance function is not differentiable, and if it is differentiable, its gradient is not explicitly computable.

The Nelder and Mead Simplex algorithm [61] is one of the most commonly used direct search methods. In fact, prior to this thesis, the Simplex algorithm was used in the Active Harmony system [23]. Despite its popularity, the Simplex algorithm has several shortcomings, which motivated us to consider Rank Ordering algorithms. The Rank Ordering algorithms are an alternative group of direct search algorithms. In the following we briefly review the Simplex and Rank Ordering algorithms, and then describe our Parallel Rank Ordering implementation for parameter tuning.

3.2.1 Simplex Algorithm

The Nelder-Mead Simplex method is a direct search algorithm for minimizing a function of multiple variables. For a function of N variables, the algorithm maintains a set of $N+1$ points forming the vertices of a simplex or polytope in N-dimensional space. This simplex is successively updated at each iteration by discarding the vertex having the highest function value and replacing it with a new vertex having a lower function value.

Let $v_0, \cdots, v_N$ be the vertices of the corresponding simplex and v_N be the point with the highest function value among them. We first compute c, the centroid of the other simplex vertices:

$$c = \left(\sum_{i=0}^{N-1} v_i \right) / N \tag{3.3}$$

The point v_N will be replaced by a point on the line $v_N + \alpha(c - v_N)$. Typically, $\alpha \in \{0.5, 2, 3\}$, but the α selection process depends on implementation. Usually, the first step is to compute the function value at the reflection point ($\alpha = 2$). Depending on the result we either check for expansion ($\alpha = 3$) or contraction ($\alpha = 0.5$). If none of the computed values is less than the function value at v_N, we contract the whole simplex around the best point.

The simplex algorithm works well, however, the algorithm is inherently sequential and is not able to fully take advantage of parallel infrastructures. In the online tuning applications that we consider, often there are multiple processors that are executing the same or very similar code and after each iteration they exchange information. Therefore, it is desirable to use different parameter values on different processors and evaluate their performance concurrently. Rank Ordering algorithms can take advantage of the concurrent performance evaluation to speedup convergence.

3.2.2 Rank Ordering Algorithm for Parameter Tuning

Kolda, et al. [47] introduces a class of reliable direct search algorithms, Generating Set Search (GSS). They also prove that if the objective function f is continuously differentiable, then GSS produces a sequence of points X_k such that,

$$\lim_{k\to\infty} \inf \|\nabla f(X_k)\| = 0. \tag{3.4}$$

This result is similar to the convergence results for gradient based algorithms,

Algorithm 1 : Sequential Rank Ordering

```
1: Start with initial simplex with vertices $\{v_0^0, \cdots, v_0^n\}$ and evaluate $f(v_0^j), \quad j = 0, \cdots, n$
2: $k = 0$
3: while Stopping criteria not valid do
4:   Reorder simplex vertices so that $f(v_k^0) \leq \cdots \leq f(v_k^n)$
5:   $r_k = 2v_k^0 - v_k^n$, evaluate $f(r_k)$ {Reflection checking step}
6:   if $f(r_k) < f(v_k^0)$ then
7:     $e_k = 3v_k^0 - 2v_k^n$, evaluate $f(e_k)$ {Expansion checking step}
8:     if $f(e_k) < f(r_k)$ then {Accept expansion}
9:       $v_{k+1}^j = 3v_k^0 - 2v_k^j$, evaluate $f(v_{k+1}^j) \quad j = 1, \cdots, n$ {Expansion steps}
10:    else {Accept reflection}
11:      $v_{k+1}^j = 2v_k^0 - v_k^j$, evaluate $f(v_{k+1}^j) \quad j = 1, \cdots, n$ {Reflection steps}
12:    end if
13:  else {Accept shrink}
14:    $v_{k+1}^j = 0.5(v_k^0 + v_k^j)$, evaluate $f(v_{k+1}^j) \quad j = 1, \cdots, n$ {Shrink steps}
15:  end if
16:  k = k+1
17: end while
```

since it guarantees that the GSS algorithms converge to a stable point of the objective function f. Therefore, compared to non-GSS direct search methods, the GSS algorithms have predictable and more reliable performance.

In Active Harmony, we use Rank Ordering direct search algorithms [53], which are in the GSS class and constitute all necessary conditions for convergence. The Rank ordering algorithms can leverage parallelism [25] to speedup convergence. For clarity and brevity, we explain Rank Ordering algorithms in the basic form appropriate for our application and not in the most general form.

Rank Ordering algorithms start with an initial simplex with vertices $\{v_0^0, \cdots, v_0^n\}$ and evaluate f at all vertices. The initial simplex should span the optimization space, hence $n \geq N$. At every iteration, the simplex is either reflected, expanded or shrunk around its best vertex (the vertex with the least function value).

Algorithm 1 is the basic Sequential Rank Ordering (SRO) algorithm given in

[53]. At each iteration, one of the three possible steps of reflection, expansion, or shrink will be accepted. Examples for each one of these steps are shown in figure 3.3 for a 2-dimensional space and a 4 point simplex.

At each iteration r_k, the reflection point of the worst vertex, v_k^n around the best vertex, v_k^0 is computed. Intuitively, the direction from the point with lowest function value to the point with highest function value approximates the gradient direction. If $f(r_k)$ is less than $f(v_k^0)$, i.e., the *best* performance point on the simplex, we compute e_k, the expansion of the worst point. If the expansion point performance is better than the reflection point, we accept the expansion, otherwise we accept the reflection. Note that reflection and expansion are only accepted if their performance is better than the *best* point discovered so far (otherwise we shrink the simplex). This is different from the Simplex algorithm approach, where the reflection is accepted if its performance is better than the *worst* vertex of the simplex.

The Parallel Rank Ordering (PRO) for parameter tuning is given in Algorithm 2. Function $\Pi(.)$ that is used in PRO description is projection mapping — its purpose is to make sure that the computed points always belong to the admissible region (i.e. the points satisfy the constraints, see section 3.2.3). After initialization steps the main loop starts in line 3. In the main loop, performance (function value) at all the reflection points are found in parallel on n processors (line 5). If reflection is successful, which means that there is at least one reflected point with better performance than the best point of the simplex, we check for expansion (line 8). Recall that in the SRO, we only compute one point in the reflection checking step (line 5 of Algorithm 1). Therefore, the PRO criteria for reflection and expansion, since it

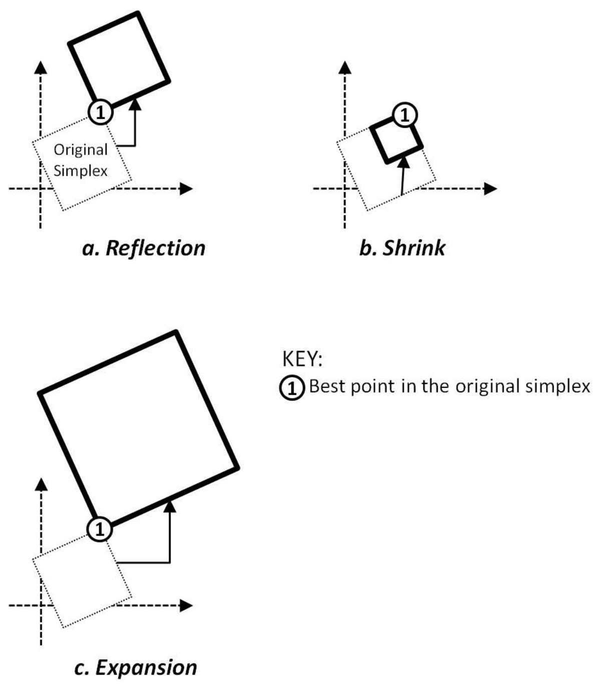


Figure 3.3: Original 4 point simplex in a 2-dimensional space, along with the simplex transformation steps

relies on performance of n points, is more reliable and improves the performance.

Note that before computing all expansion points (line 10), we check the outcome of the expansion for only the most promising case first (line 8). The most promising point (shown in Figure 3.4) is the point in the original simplex whose reflection around the best point returns a better function value. This seems to be

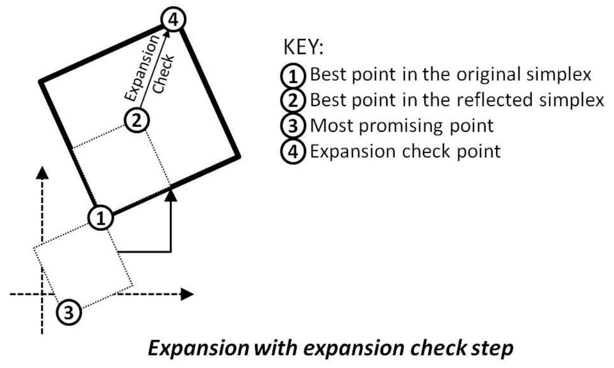

Expansion Check
KEY:
1 Best point in the original simplex
2 Best point in the reflected simplex
3 Most promising point
4 Expansion check point
Expansion with expansion check step

Figure 3.4: Illustration for expansion check step

Algorithm 2 : Parallel Rank Ordering

```
1: Start with initial simplex with vertices $\{v_0^0, \cdots, v_0^n\}$ and evaluate $f(v_0^j), \quad j = 0, \cdots, n$
   in parallel on $n$ processor.
2: $k = 0$
3: **while** Stopping Criteria Not Valid **do**
4:    Reorder simplex vertices, so that $f(v_k^0) \leq \cdots \leq f(v_k^n)$
5:    Compute $n$ reflection points $r_k^j = \Pi\left(2v_k^0 - v_k^j\right)$, and function values $f(r_k^j), \quad j =$
      $1, \cdots, n$ in parallel on $n$ processors. $\{$*Reflection step*$\}$
6:    $l = \arg\min_j f(r_k^j)$
7:    **if** $f(r_k^l) < f(v_k^0)$ **then**
8:       $e_k = \Pi\left(3v_k^0 - 2v_k^l\right)$, evaluate $f(e_k)$ $\{$*Expansion checking step*$\}$
9:       **if** $f(e_k) < f(r_k^l)$ **then** $\{$*Accept expansion*$\}$
10:         Compute $n$ expansion points $e_k^j = \Pi\left(3v_k^0 - v_k^j\right)$, and function values
            $f(e_k^j), \quad j = 1, \cdots, n$ in parallel on $n$ processors. $\{$*Expansion step*$\}$
11:         $v_{k+1}^j = e_k^j \quad j = 1, \cdots, n$
12:      **else** $\{$*Accept reflection*$\}$
13:         $v_{k+1}^j = r_k^j \quad j = 1, \cdots, n$
14:      **end if**
15:   **else** $\{$*Accept shrink*$\}$
16:      Compute $\Pi\left(v_{k+1}^j = 0.5v_k^0 + 0.5v_k^j\right)$, and $f(v_{k+1}^j) \quad j = 1, \cdots, n$ in parallel on $n$
         processor. $\{$*Shrink step*$\}$
17:   **end if**
18:   k = k+1
19: **end while**
```

counter-intuitive at first glance, since we are not taking full advantage of the parallelism. However, in our experiments, we realized there are some expansion points with very poor performance that can slow down the algorithm (for example, setting a tile size to zero). Therefore, to avoid these time consuming instances and to ensure good transient behavior of the search algorithm, we calculate the expansion point performance for the most promising case first and only if it is successful, perform a full expansion of the simplex.

If reflection is not successful and there is no reflected point performing better than the best point of the simplex, shrinking of the simplex is accepted. All the shrinking points and their performance are computed in parallel. Each iteration of the PRO algorithm, thus, takes at most 3 timesteps (reflection, expansion checking, and expansion steps). In the following, we will go over some PRO specific implementation issues.

3.2.3 Projection Operator

Parameter tuning is a constrained optimization problem. Therefore, in each step we have to make sure that the points computed by simplex transformation steps are admissible, i.e. they satisfy the constraints. The projection operator $\Pi(.)$ takes care of this problem by mapping points that are not admissible to admissible points. We consider two types of parameter constraints: boundary constraints and internal discontinuity constraints.

Boundary constraints are upper and/or lower limits for the parameters. If the computed value for a parameter is less (greater) than the lower (upper) limit, the projected value for that parameter would be equal to the lower (upper) limit.

Some tuning parameters can only have admissible discrete values. For instance, many of the variables are finite integer numbers. The projection operator makes sure that the computed parameters are rounded to an admissible discrete value. Consider the point $x = (x(1), \cdots, x(N)) \in R^N$ that is computed after a transformation (reflection, extraction, or shrink) around the point v_k^0 in PRO. For every parameter i, if $x(i)$ is admissible it will remain the same. Otherwise, if $l(i) < x(i) < u(i)$, for two consecutive admissible values $l(i)$ and $u(i)$, then projection of $x(i)$ is $l(i)$ if $v_k^0(i) < x(i)$, and is $u(i)$ if $v_k^0(i) > x(i)$. In other words, every parameter is rounded to its lower or higher discrete value, whichever is closer to the transformation center $v_k^0(i)$. In this way, after a finite number of consecutive shrinking transformations, all discrete parameters $x(k)$ become equal to $v_k^0(i)$. This property will be used to check convergence in the stopping criteria.

This aforementioned methods work well for hyper-rectangular search spaces, but not when we have an arbitrarily shaped space defined by (possibly non-linear) constraints on parameter values. To account for arbitrarily shaped spaces, we provide two implementations of the projection operator:

1. *L_1 distance based method:* In this method, we geometrically project an inadmissible point to its nearest neighbor. We define distance between two points using L_1 distance, which is the sum of the absolute differences of their coordinates. The nearest neighbor of an inadmissible point (calculated in terms of L_1) will thus be a legal point with the least amount of *change* (in terms of parameter values) summed over all dimensions.

   Computing the least L_1 distance unfortunately involves finding the nearest neighbors in a high dimensional space, which is a computationally intensive task. After experimenting with multiple nearest-neighbor algorithms, we adopted the Approximate Nearest Neighbor (ANN) [5] algorithm for two reasons. First, for approximate neighbors, ANN has linear space requirements and logarithmic time complexity on the number of points in the search space. Second, an efficient implementation of the ANN library is available [58]. The library supports a variety of metrics to define distance between two points, including L_1 distance metric. We elaborate more on how we actually implemented this projection method in chapter 4.

2. *Penalty Factor:* The aforementioned distance-based method is computationally intensive and may slow down online tuning. Thus, for online tuning, we use a penalization method to handle boundary constraints. We add a penalty factor to the performance metric associated with the points that violate the constraints. The idea is to discourage the simplex from moving towards illegal regions of the search space. This approach has been used previously in the context of constrained optimization using genetic algorithms [40]. In all of the online tuning experimental results presented in chapter 8, we use this method because it is simple and light-weight.

3.2.4 Simplex Construction and Size

The initial simplex, with size kN, needs to be non-degenerate so that it can span the whole parameter space; therefore, kN must be at least $N + 1$, where N is the number of tunable parameters. To exploit all available parallelism, kN can be set to the number of resources/processors available.

We provide a set of initial simplex construction methods. Applications can choose to use a particular method at the start of a tuning session. These methods range from a completely random simplex to user-defined set of simplex points. Here we describe the most commonly used method. We start by randomly selecting kN

points[2]. The first iteration of the algorithm evaluate these random configurations. The initial simplex is then constructed by randomly sampling points at distance d (L_1 distance) from the best performing point. The distance d is problem dependent and end-users can specify this as a parameter to our algorithm. The set of search directions/vectors (from the initial best point to the sampled points) generated in this fashion is guaranteed to be a linearly independent set, which in turn guarantees that the simplex points span the search space.

3.2.5 Stopping Criteria

After every iteration, the algorithm checks to see if all simplex vertices are the same (since we deal exclusively with discrete parameters). If that is the case, the search algorithm has converged to a point in the search space. At this stage, we construct a new simplex by randomly sampling points within d (L_1 distance) from the convergence point. We evaluate the new set of points in parallel; if none of them outperforms v_k^0, then v_k^0 is a local minimum and we can stop, otherwise we can continue PRO with the generated simplex.

Applications can specify the number of new-direction trials. We start d at one and keep increasing the distance until either one of the newly computed points is better than v_k^0 or we reach the maximum number of trials specified by the application.

[2]Note that this randomized method can sometimes select points in the search space that have poor performance. We pay this penalty for one iteration at the beginning of the search to gather some knowledge about the search space. The cost of this step is usually amortized within the first few PRO steps.

3.3 Performance Variability and its Impact on Parameter Tuning

Besides the tunable parameters, there are many other factors affecting a program's performance. Therefore, even for a fixed set of tunable parameters, the application performance varies in time. Other applications running on the same processor, network contention, operating system, and memory architecture are common sources of performance variability. In this section, we provide a simple stochastic model for performance variability and present evidence, based on measurements from a real cluster, indicating the presence of a *heavy-tail* component in the probability distribution function (PDF) associated with the application execution time data.

3.3.1 Two Job Model

We model the computing system as a single machine with a strict priority scheduler serving two sets of jobs. The tunable application is the second priority job and all sources of performance variability are modeled as first priority jobs (i.e. background workload). The computing system serves the application, when there are no first priority jobs in the system. First priority jobs arrival and service time are random processes; therefore, the application performance (finishing time of the second priority job) is a random variable (r.v.).

Let $f(v)$ be the *ideal* application execution time for parameter value v, when there are no first priority jobs in the system. The *real* (or *observed*) application performance, y, when there are background jobs in the system is:

$$y = f(v) + n(\cdot). \tag{3.5}$$

The r.v. $n(\cdot)$ is the time the system spends serving first priority jobs while the application is in the system. It will shortly become clear that, even in this simple model, $n(\cdot)$ is a function of v. Hence, in our analysis, we cannot assume that noise (serving time of first priority job) is independent of the selected parameter values v.

Let ρ be the fraction of time the system spends, on average, to process the first priority jobs. Let $E(y)$ be the average *observed* application execution time. With these assumptions, the average time that the system spends serving first priority jobs while the application is waiting is $\rho E(y)$, and $(1-\rho)E(y)$ is the time that the system spends serving the application. But the serving time of application (excluding the waiting time) is, by definition, $f(v)$[3]. Hence, the average *observed* execution time of the application is:

$$E(y) = \frac{f(v)}{1-\rho}. \tag{3.6}$$

Taking the average of (3.5), we have,

$$E(y) = f(v) + E(n(\cdot)). \tag{3.7}$$

[3]Note that $f(v)$ is a deterministic function - when there are no background jobs in the system, the *ideal* application performance (for parameter value v) is constant.

From equations (3.6)-(3.7) we have:

$$E(n(\cdot)) = \frac{\rho}{1-\rho} f(v). \tag{3.8}$$

Now, it should be clear that the expected variability, $E(n(\cdot))$, is a linear function of $f(v)$; hence, the r.v. $n(\cdot)$ is a function of the application parameters v and we can write

$$y = f(v) + n(v). \tag{3.9}$$

3.3.2 Heavy-tail Model

In the previous section, using the two job model, we showed that the expected performance is a function of ρ, background workload level. In this section, we attempt to capture the characteristics of performance variability that are critical for the optimization process.

Previous studies of the performance variability indicate that there is a *non-negligible* probability of observing large variations in the finishing time of an application [49, 69]. When application execution time is measured at per timestep granularity, large variations in the finishing time can be mainly attributed to few timesteps that take relatively long time to complete. This behavior can be characterized through the use of *heavy-tail* models. Heavy-tail distributions exhibit tails that decay as a hyperbolic function, which is in contrast to the typical exponential decay in other models such as a Gaussian distribution.

A distribution is said to have a heavy-tail if:

$$P[X > x] \sim x^{-\alpha}, \text{as } x \to \infty, \quad 0 < \alpha < 2 \tag{3.10}$$

This means that regardless of the distribution for small values of the random variable, if the asymptotic shape of the distribution is hyperbolic, it is heavy-tailed [21]. The simplest heavy-tailed distribution is the Pareto distribution which is hyperbolic over its entire range and its cumulative distribution function (cdf) is given by:

$$F_X(x) = P[X \leq x] = 1 - (\beta/x)^{\alpha}, \tag{3.11}$$

where β is the smallest value the X can take. For $1 < \alpha < 2$, Pareto distribution has finite mean and infinite variance, and for $0 < \alpha \leq 1$, both mean and variance are infinite.

Heavy-tailed distributions have properties that are qualitatively different from commonly used distributions such as Exponential or Poisson distributions. Therefore, it is important to know if the performance variability distribution is heavy-tail. In the next section, we will try to answer this question using GS2 as our subject parallel application.

3.3.3 Case study: GS2

We use the GS2 [26, 48] application to study the performance variability, when the application parameters are fixed. GS2 has several tunable parameters, which can be set to represent the appropriate conditions for different modes.

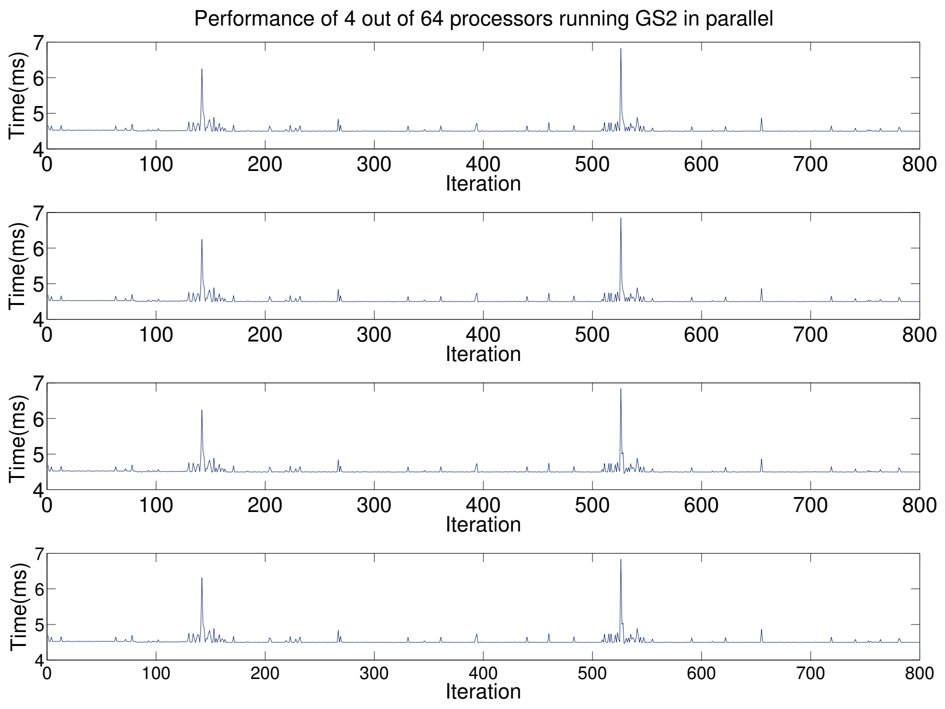


Figure 3.5: Running time for 800 iterations of the GS2 program on 4 out of 64 parallel processors

Figure 3.5 shows the running time of the GS2 with *fixed* parameters for 800 timesteps on 4 processors from a 64 processor run[4]. Clearly, there are two distinct types of spikes (or equivalently, timesteps that take relatively long time to complete) in the plots: big and small. There is also high correlation and similarity between the curves. Regardless of the cross-processor correlation, the existence of spikes is evidence of a heavy-tail component.

We do not know if the source of the observed variation in execution time between time steps is due to the application, or due to the system it runs on.

[4]Runs were conducted on a 64 node Linux cluster. More information on the cluster is provided in section 6.4.

However, for the purposes of designing a robust optimization algorithm, the source of this variability is not important, but its properties (i.e. is it heavy-tailed) is what matters.

We use two methods to detect heavy-tailed behavior in GS2 execution time dataset. In the first method, we draw a log-log plot of complementary cumulative distribution frequency, $(1 - cdf)$, which is $P[X > x]$. For the heavy-tail r.v., the tail of the log-log plot should be approximately linear. The second method estimates the tail index, α in (3.10), using the method[5] suggested by Crovella et al [22]. We should note that while these methods are useful in detecting heavy-tailed behavior in a dataset, they can only *suggest* that such behavior is present; they cannot conclusively confirm heavy-tailed property.

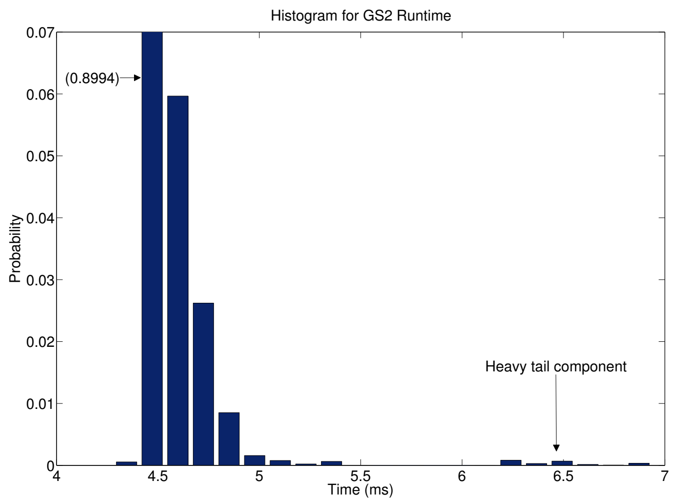


Figure 3.6: pdf of the GS2 data

Figure 3.6 is the PDF of all 64 processors performance data. As we expect, the last six bars are not negligible. Figure 3.7 is the $(1 - cdf)$ log-log plot for the

[5]An efficient C-implementation of the method is available at `http://www.cs.bu.edu/~crovella/aest.html`. For our estimation of α, we used author suggested values for the aggregation factor (2) and aggregation levels (10).

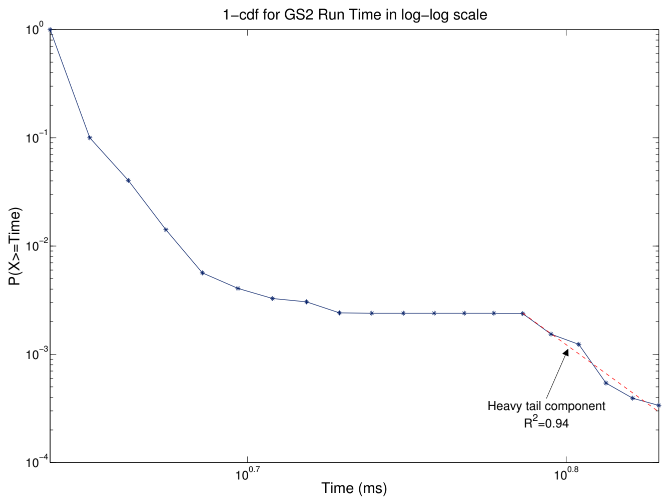


Figure 3.7: 1-cdf of the GS2 data

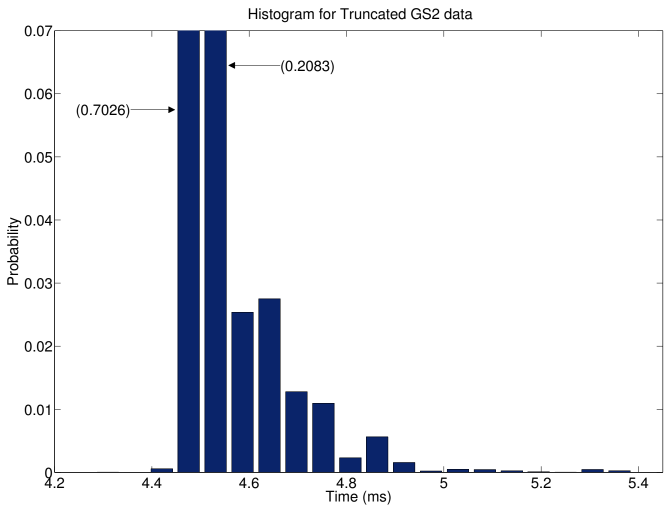


Figure 3.8: pdf of the truncated GS2 data

same GS2 data and the last part (tail) of the graph approximately forms a line. A least squares linear fit (with $R^2 = 0.94$) to the tail is shown in the figure. Using the method suggested by Crovella et al [22], we estimated the tail index, α, in relation (3.10) to be 1.21, which indicates that the distribution has finite mean and infinite variance.

In order to study characteristic of the small spikes in figure 3.5, we truncate the GS2 data and remove all samples that are larger than 5.5. The pdf and $(1 - cdf)$

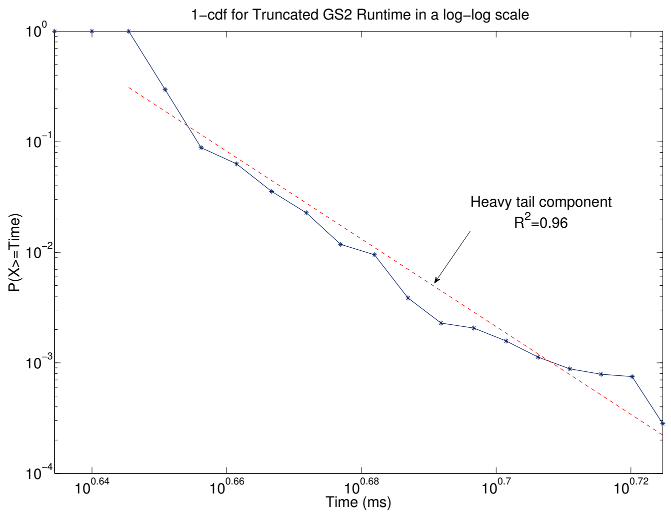


Figure 3.9: 1-cdf of the truncated GS2 data

plots for the truncated data are shown in figure 3.8 and 3.9 respectively. Evidence for a heavy-tail component, which is due to the small spikes this time, is present in the plots. We estimated the tail index, α, to be 1.37 for the truncated dataset.

In summary, the data presented in this section suggests that the performance variability is heavy-tailed. What this means from the perspective of auto-tuning is that there is a non-negligible probability of observing large variations in the measured performance. The sampled performance measurements could, therefore, have infinite variance. This observation essentially renders average operator ineffective. Average is the most widely used operator to aggregate and estimate "real" performance from multiple samples. As an alternative, we showed, in our earlier work [83], taking a minimum of multiple performance measurements is an effective way for performance estimation even in the presence of heavy-tail component in the performance distribution. The minimum has finite mean and variance and is not heavy-tailed. We note even in the presence of 5% variability due to background noise, taking the minimum of two samples is enough to ensure the convergence of the search algorithm.

3.4 Summary

In this chapter, we presented the Parallel Rank Ordering algorithm for tuning of application parameters. The algorithm is well-suited for navigating parameter spaces for parallel applications because it can leverage parallelism to search for parameters in parallel. We discussed various implementation-level aspects of the algorithm and provided methods to prevent the search algorithm from venturing into inadmissible regions of the search-space. We also studied the nature of performance variability and showed results that indicate the performance variability of applications on clusters is heavy-tailed.

Chapter 4

The Framework — Active Harmony

In this chapter, we describe our search-based auto-tuner — Active Harmony. The goal of our framework is to bring together all tunable targets in a given application within a single unified auto-tuning framework. Entities within an application (e.g. library parameters, computationally intensive loop nests) that can be changed or transformed (for better performance) without affecting the application result are appropriate for exploration by Active Harmony.

Active Harmony takes a search-based collaborative approach to auto-tuning. Our system allows application programmers, library writers, compilers and performance modelers to describe and export a set of performance related tunable parameters. These parameters define a tuning search-space. More often than not, this search-space is high-dimensional and exponential in size and thus, cannot be explored manually or even exhaustively with automation. Our system monitors the program performance and makes adaptation decisions. The decisions are made by a central controller using a parallel search algorithm[1]. The parallel search algorithm leverages parallel architectures to search across a set of optimization parameter values. Different nodes of a parallel system evaluate different configurations at each

[1]In addition to the parallel search algorithm, Active Harmony provides a selection of other algorithms — Nelder-Mead simplex algorithm, Brute-force algorithm and Random search algorithm. The search algorithm choice can be made at Active Harmony server launch time. Unless otherwise indicated, the auto-tuning experiments reported in this document were done using the parallel search algorithm.

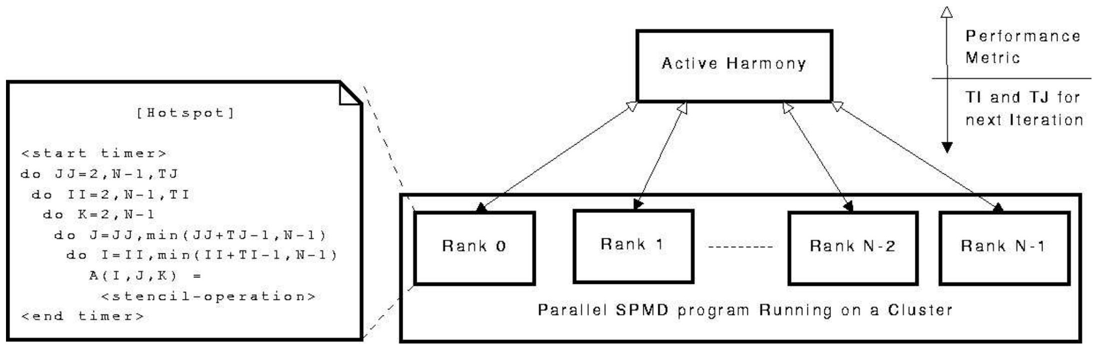
timestep.

The design of Active Harmony was originally proposed by Hollingsworth and Keleher [36]. Since then, the software architecture has evolved along several dimensions. The current design uses the client-server model. Figure 4.1 shows a simple use-case scenario for Active Harmony. The example shows a "harmonized" (see section 4.1) SPMD-based parallel application[2]. Lets assume the computational hotspot for this application is the Jacobi kernel (a stencil code). A small fragment of the tiled three-dimensional Jacobi kernel is shown in the figure. Active Harmony is used to conduct the search for tile-control variables — `TI` and `TJ`. For SPMD-based programs, each MPI process acts as an independent Active Harmony client. Thus, with the use of the Parallel Rank Ordering(PRO) as the search algorithm, we can have different processors evaluate completely different values for the tunable parameters (`TI` and `TJ` in the figure 4.1) at each timestep. Each client reports the performance measurement corresponding to the values of the tunable parameters it received to the Active Harmony server. The server bases its adaptation decisions on this performance data.

4.1 Harmonization

For online tuning, developers can use the Active Harmony API to add "hooks" in their application to make the application tunable. In Active Harmony vocabulary, we refer to this process of adding hooks as *harmonization*. The process involves

[2]In SPMD parallel programming model, multiple instances of the same code run on multiple processors.


Active Harmony
Performance Metric
TI and TJ for next Iteration
Rank 0
Rank 1
Rank N-2
Rank N-1
Parallel SPMD program Running on a Cluster

```
           [Hotspot]

<start timer>
do JJ=2,N-1,TJ
 do II=2,N-1,TI
  do K=2,N-1
   do J=JJ,min(JJ+TJ-1,N-1)
    do I=II,min(II+TI-1,N-1)
      A(I,J,K) =
       <stencil-operation>
<end timer>
```

Figure 4.1: A simple use-case scenario for Active Harmony

making fairly small changes to the application code to call library routines that make a connection to the Active Harmony server, export tuning options, update the server with performance values and import the parameter values suggested by the Active Harmony server. The phrase *harmonized application* is used to refer to an application that uses Active Harmony to adapt its execution.

For offline-tuning, we provide a set of driver programs that end-users can use to drive the tuning process by repeatedly invoking the application with different command line options or input files. We elaborate more on these driver programs in chapter 7.

4.2 Need to Coordinate Auto-tuners

Since auto-tuning can take place at many levels of a program from the user to specific libraries, an important question is how to coordinate this process. Left uncoordinated, each component of a program may try to run its own auto-tuner. Such a process would likely lead to a dissonance where multiple components change something nearly simultaneously and then try to assess if it improved the program's performance. However, without coordination it would be impossible to tell which change was actually improving the program. In fact, it is likely that one change might improve performance and the second hurt performance and the net performance benefit is little or none. Thus an important question is how to best coordinate the efforts of auto-tuners. There are a variety of approaches possible ranging from simple arbitration to ensure that only one auto-tuner is running at once to a fully unified system that allows coordinated simultaneous search of parameters originating from different auto-tuners.

In the Active Harmony project, we distinguish between dependent and independent auto-tuning targets and use the following coordination methods.

1. *Dependent auto-tuning targets:* For dependent auto-tuning targets (for example, multiple code-sections within a single function (or even a single loop) that share data-structures), we take the approach of a coordinated system to allow all auto-tuning targets to be tuned together. We accomplish this by having each target expose its tunable parameters. A central search engine then manages the evaluation of possible auto-tuning steps. The core search algorithm is able to decide which sources of auto-tuning information should be considered and when.

2. *Independent auto-tuning targets:* For independent auto-tuning targets (for example, code-sections in different libraries used by the application that do not affect each other in terms of the application execution and data structures), application developers can choose to start multiple (and parallel) auto-tuning

sessions with separate Active Harmony servers to allow simultaneous tuning of multiple targets. Developers can also choose to use different search algorithms for different targets.

4.3 Collaborative Tuning via CSL

We firmly believe that the success of any high performance computing research depends on how effectively application developers, domain-experts and auto-tuners communicate and work together. However, to our dismay, we realized that there is a severe communication gap that exists between these communities. Even when so many auto-tuning frameworks are available, the use of these frameworks has not yet been a part of the mainstream application development. We believe that the hesitancy in part of the developers to use the tools is mainly due to the lack of a simple and common interface that can be used to export their tuning needs to the auto-tuners.

Search-based auto-tuners require a precise specification of valid parameters. Such specification could be as simple as expressing the minimum, maximum and initial values for a parameter. Sometimes not all parameters values within a range should be searched, so option to specify a step function is useful. Likewise for parameters with a large range (i.e. a buffer that could be from 1K to 100Megabytes), it is useful to specify that parameter should be searched based on the log of the value.

Another critical factor is that not all parameters are independent. Frequently, there is a relationship between parameters (i.e. when considering tiling a two dimensional array, it is often useful to have the tiles be rectangles with a definite aspect ratio). To meet the needs of having a fully expressive way for developers to define parameters, we have developed a simple and extensible language (Constraint Specification Language, CSL) that standardizes parameter space representation for search-based auto-tuners. CSL allows tool developers to share information and search strategies with each other. Meanwhile, application developers can use CSL to export their tuning needs to auto-tuning tools.

CSL provides constructs to define tunable parameters and to express relationships between those parameters. Dependencies between different parameters can be easily specified using mathematical expressions. CSL supports a fairly comprehensive list of mathematical, logical and relational operators. Hints to the underlying search algorithm in the form of initial points to start the search, default values for parameters, simple constraints on parameters such as MPI message-sizes, number of OpenMP threads, etc. can be easily expressed using CSL. Information from performance models can be specified in the form of constraints to guide the search. Furthermore, parameters can also be grouped into different categories to allow application of similar tuning strategies. This is particularly helpful when there are multiple code-sections that benefit from the same optimization. We provide the full CSL grammar in tables A.1 and A.2 (in the Appendix).

We have developed and released a standalone tool that takes a CSL description and checks for the correctness of the description. The underlying parsing framework is ANTLRv3 [68]. The tool also generates a python script based on the parameters and constraints specifications provided in the description. This python script uses

Table 4.1: A simple CSL specification and the corresponding python output

CSL Description

```
search space simple {
     # parameter definitions
     parameter x int {
       range [1:8:1];
     }
     parameter y int {
       range [1:8:1];
     }
     parameter z int {
       range [1:8:1];
     }
     # constraints
     constraint cone {
       x+z>=z;
     }
     constraint ctwo {
       y>z;
     }
     #putting everything together
     specification {
       cone && ctwo;
     }
}
```

Python output

```
from constraint import *

simple=Problem()
# parameter declarations
class x:
   def values(self):
       ls=[1,2,3,4,5,6,7,8]
       return ls
simple.addVariable("x",x().values())
class y:
   def values(self):
       ls=[1,2,3,4,5,6,7,8]
       return ls
simple.addVariable("y",y().values())
class z:
   def values(self):
       ls=[1,2,3,4,5,6,7,8]
       return ls
simple.addVariable("z",z().values())

# Constraints
def cone (z,x):
   return ((x + z) >= z)
def ctwo (z,y):
   return (y > z)
def specification (z,y,x):
   return (cone(z, x) and ctwo(z, y))
simple.addConstraint(FunctionConstraint(\
  specification), ("z","y","x"))

sols=simple.getSolutions()
```

the python-constraint module [64], which offers solvers for Constraint Solving Problems (CSPs) over finite domains. The users can then run the python interpreter on the generated script to generate all valid points in the search space. These points are used by the projection server (discussed in the next section) to make projection decisions.

We provide a simple parameter specification example in Table 4.1. In this example, the search space consists of three parameters — `x`, `y` and `z`. The relationships between these parameters are expressed using two constraint definitions — `cone` and `ctwo`. We also provide the python script output generated by our standalone tool in Table 4.1. In the appendix, we provide an example of a more elaborate parameter specification using the CSL (in Table A.3). This specification defines tiling and unrolling parameters for matrix multiplication tuning. It also defines constraint relations between the parameters. We provide the python script output generated by our standalone tool in Table A.4 (in the Appendix).

4.4 Framework Components

Apart from the Active Harmony server, which consists of the optimization backend implemented in Tcl/Tk, the auto-tuning system consists of three other main components — code-server, projection server and database.

4.4.1 Code-server

Our system relies on code-server to generate and compile code for tunable parameters that require new code to move from one admissible value to another. Code-server is a distributed code-generation and compilation tool. Users provide a list of machines at the start of the auto-tuning session. Upon receiving the request to generate code for a set of parameters, the task is farmed out to the machines specified by the user. We describe the design of code-server in great details in chapters 6 and 8.

4.4.2 Projection Component

The design of this component is based on the client-server model and the component consists of a projection server and a client API. The projection server is used by the optimization kernel of Active Harmony to “project” inadmissible points (points that violate CSL constraints) in the simplex to the admissible region in the search space. The optimization kernel uses the projection API function calls to connect and communicate with a projection server. The projection API is listed in Table 4.2. Since the optimization backend of Active Harmony is implemented using Tcl/Tk, a tcl-callable API is generated using swig [82].

The projection server returns the nearest L_1 neighbor (in the admissible region) for a given inadmissible point. The projection server relies on ANN[5, 58] to do this nearest neighbor calculation. Using the standalone tool that we described in section 4.3, users first convert the parameter space definition (written in CSL) into a python script. The python interpreter then enumerates all legal points in the parameter space. The data file which consists of all possible points is read by the projection server to populate the underlying data-structures. This operation can be expensive if the size and the dimension of the search space is large. However, this is a one time process that happens at projection server startup. When the Active Harmony search algorithm generates new points via simplex transformations, a request to the projection server is made to verify that the points are valid. For each invalid point, the projection server returns its nearest neighbor in the legal region.

4.4.3 Database Component

It is evident from the discussion of the stopping criteria (in chapter 3) that as the simplex nears convergence, multiple vertices in the simplex become identical. We

Table 4.2: Projection API

```
/*
 * Connecting to a projection server.
 */
int projection_startup(char* hostname, int sport);

/*
 * Announce an end of the session to the server.
 */
void projection_end();

/*
 * Is given point admissible?
*/
int is_a_valid_point(char* point);

/*
 * initial simplex construction based on a given initial point.
 */
void simplex_construction(char* request);

/*
 * Project a point.
 */
char* do_projection_one_point(char *request);

/*
 * Project a set of points.
 */
char* do_projection_entire_simplex(char *request);
```

exploit this property of the search algorithm to reduce the offline tuning time. The Active Harmony server maintains a global database of the candidate configurations and the associated performance metric. Clients can query the server to see if the configuration that they have been assigned has been evaluated earlier (possibly by other clients). If the server does find an entry, it notifies the client and the client can choose not to run the application to save tuning time[3].

For online tuning, the database component can be used to distribute the best configuration to all the clients at periodic intervals.

[3]We realize that instead of the clients asking the server, we can have the server send an "already evaluated" message. This is an implementation-level detail that we missed.

4.5 Summary

In this chapter, we presented our auto-tuning framework. We discussed various components of the system — code-server, projection server and the database component. We elaborated on how these components interact with each during an auto-tuning session. We also introduced CSL, which is designed to promote greater collaboration between application developers, compilers and auto-tuners.

Chapter 5

Application-level Auto-tuning

Scientific applications and libraries often include tunable input parameters that users can select at launch time to optimize the application's performance. These input parameters are meant to control several important aspects of the application performance such as data decomposition and alignment, numerical algorithm selection, communication protocol selection, etc. Choosing appropriate values for these parameters is essential in getting maximum application throughput. Figure 5.1 underscores this fact. The figure shows the performance (application execution time) of GS2 [26, 48], a physics application developed to study low-frequency turbulence in magnetized plasma, as a function of two input parameters. All other input parameters are fixed. Clearly, the parameter space is not smooth and contains multiple local minimums. The best and worst points are a factor of 10 different.

The task of making a good selection of the input parameters is non-trivial because this requires a concrete understanding of the interactions between the input parameters and the underlying algorithmic behaviors that they are meant to control. Moreover, the input parameters also interact with the elements of the target architecture. Compounding these challenges is the fact that the input parameter space is usually high-dimensional (due to applications having many parameters) and exponential in size. Under these conditions, it is practically impossible to manually

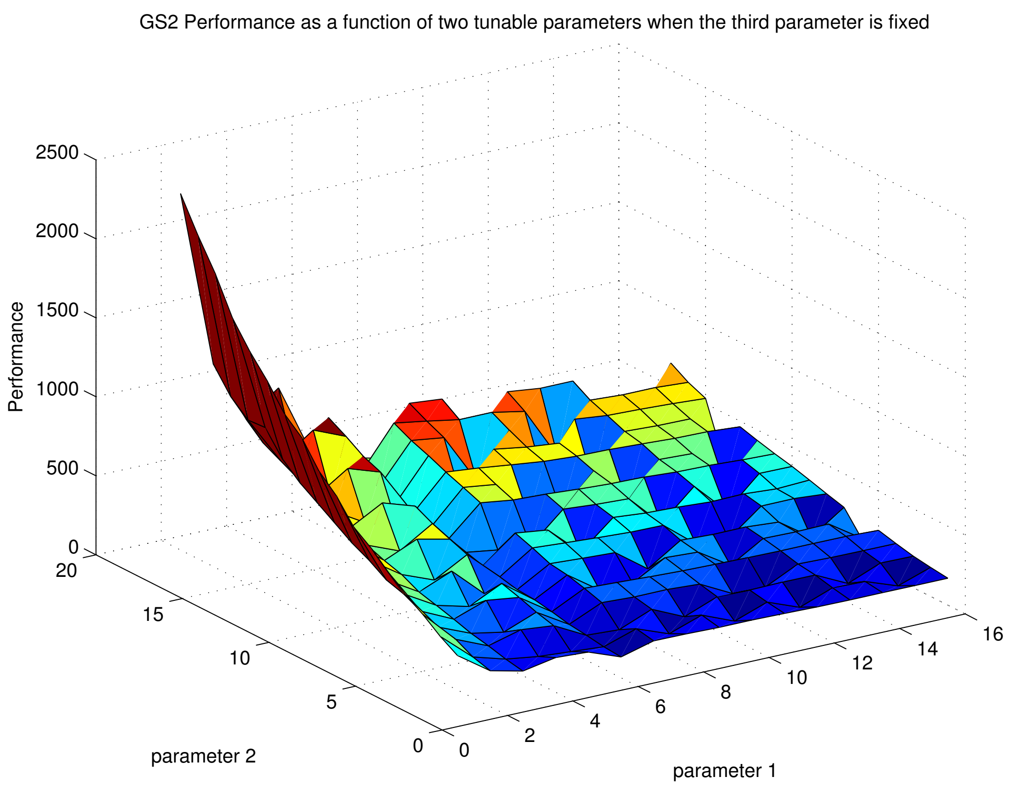


Figure 5.1: GS2 performance plot with two tunable parameters.

tune the parameters to optimize the application performance; therefore, automated parameter tuning is required.

To that end, in this chapter, we use the Active Harmony system to automatically select input parameter configurations for scientific applications. We also compare the performance of the parallel rank ordering algorithm to that of the modified Nelder-Mead simplex algorithm, a sequential algorithm. The comparison is based on the number of search iterations and the quality of the input configurations found by the two algorithms. In the next section, we describe a method that we use to establish a "pseudo"-ordering on non-numeric tunable parameters.

5.1 Parameter Ordering Scheme

Both PRO and Nelder-Mead algorithms require a rough ordering to be established among different options for a given parameter. Non-numerical parameters generally lack a natural ordering. For example, three different FFT algorithms have no natural order. Our approach to ordering these values is to measure performance by holding the rest of the parameters fixed and measuring our objective function for each value. We attempt to establish such orderings by studying the parameters in isolation. For any given parameter P, we obtain the minimum of 4 sample performance values for each of the options it can take. For each of these evaluations, we keep the problem characteristics the same and all other tunable input parameters are kept at their default values. The options are then ordered based on the performance observed during these evaluations. When selecting the order of parameters to tune, we ignore parameter interaction. However, such interaction is taken into consideration by the optimization algorithms when evaluating configurations and moving the simplexes across the parameter space.

Note that this process also gives us an additional information about which parameter to focus on and which parameter to eliminate from the search space during the optimization phase.

5.2 Empirical Results

In this section, we present our experimental results. We use Active Harmony's offline tuning mechanism to tune input parameters of three well-studied scientific application benchmarks. We compare the tuning results (in terms of the quality of parameter configurations and the number of search iterations) from PRO with the results from the Nelder-Mead simplex algorithm. When a default input configuration is available, we compare our results with default performance as well.

All experiments reported in this chapter were performed on a 64 node Linux cluster (henceforth referenced as umd-cluster). Each node is equipped with dual Intel Xeon 2.66 GHz (SSE2) processors. Nodes are connected via a Myrinet network. A PBS scheduler schedules at most one application per node at a time. L1-cache and L2-cache sizes are 128 KB and 4096 KB respectively.

Subject applications were selected from three different realms of scientific computing – nonlinear spectral method (PSTSWM), dense linear algebra (HPL) and finite difference method (POP). Each of the application has multiple tunable input parameters. The parameter space for each of the applications is fairly large.

5.2.1 PSTSWM

The main idea behind PSTSWM [91] was to embed algorithmic options into codes that allow them to be "tuned" for a particular machine without requiring code modifications. To accomplish this, the benchmark provides a variety of input parameters that can be set at program launch time to choose between several types of parallel algorithms, communication protocols and data decomposition. The code has been used extensively to evaluate performance of many high-end supercomputers [27].

The underlying numerical code of PSTSWM solves the nonlinear shallow water equations on a rotating sphere using the spectral transform method [91]. Two transformations of state variables are done at each timestep: first from the physical tensor product longitude-latitude-vertical grid to spectral domain using Fast Fourier Transform (FFT) and in the reverse direction via Legendre Transformation (LT). For both transformations, two classes of parallel algorithms are available: distributed (uses fixed data decomposition) and transpose (remaps the domain to calculate the transformations sequentially). In our study, we looked at all four combinations of these algorithms:

- Algorithm 1 (A_1): distributed FFT and transpose LT
- Algorithm 2 (A_2): distributed FFT and distributed LT
- Algorithm 3 (A_3): transpose FFT and transpose LT
- Algorithm 4 (A_4): transpose FFT and distributed LT

For each algorithm, there are 10 input parameters that can be set during launch time. The parameters, their domain-size and default values are given in Table 5.1. The *pq* parameter defines the logical processor grid. *Meshopt* determines how to map the logical processor mesh to the "physical" processors. The *commfft* (*commflt*) and *commift* (*commilt*) parameters specify which algorithm variants to be used in the parallel forward and inverse FFT (LT) algorithms respectively. The *protfft*, *protift*, *protflt* and *protilt* parameters specify the communication protocol to be used for respective parallel FFT and LT algorithms. As evident from table 5.1, the input search space is fairly large for PSTSWM - on the order 10^9. The user's manual provides some rough guidelines on how to choose the values and also provides default values. However, given the size of the parameter space, finding a good set of input configuration for a given platform requires extensive experimentation. The performance metric (objective function) chosen is the solve-time reported in PSTSWM results.

Table 5.1: PSTSWM parameters

Parameter	**Number of possible values**	**Default Value**
pq	Depends on the # of procs.	n/a
meshopt	10	1
commfft	4 (for dist. *FFT*) 12 (for trans.)	1
commift	4 (for dist. *FFT*) 12 (for trans.)	1
commflt	4 (for dist. *FFT*) 12 (for trans.)	1
commilt	4 (for dist. *FFT*) 12 (for trans.)	1
protfft	6	1
protift	6	1
protflt	6	1
protilt	6	1

For our experiments, we considered the *T85L32* problem (20 simulation hours). The problem is run on 32 processors. Algorithm A_2 had the poorest performance on our cluster. So, we exclude it from the rest of the analysis and instead focus on the results for the remaining three combinations of algorithms. Both PRO and Nelder Mead algorithms are used to search the parameter space. The performance of the best configuration is then compared with the performance of default parameter configuration.

Figure 5.2 shows the best points of the two optimization algorithms at different iterations (for A_3). The first 11 iterations for the Nelder Mead algorithm are not shown in the figure. These initial iterations are exploratory and are used to construct

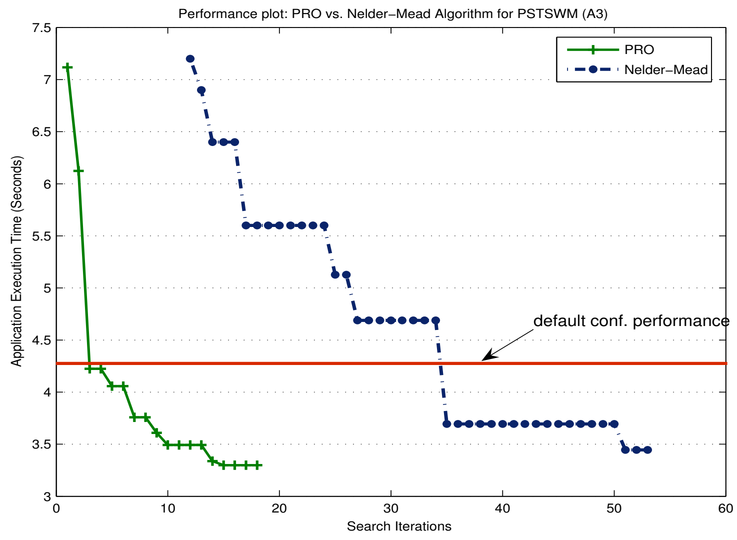


Figure 5.2: Comparison of best performing points of PRO and Nelder-Mead Algorithm

the initial $N+1$ simplex. For A_3, the default input configuration takes 4.25 seconds (minimum of four samples) to complete the *T85L32* run. PRO finds a configuration that outperforms the default execution time by 22.6% after just 15 iterations and converges to a point in the 18th iteration. Nelder Mead finds a configuration which outperforms default configuration after 35 iterations. However, it took 54 iterations for the Nelder Mead algorithm to find an input set that outperforms the default configuration by 19%.

Table 5.2: Summary of results for PSTSWM

Algorithm	**Application Runtime (sec)**			**Number of Iterations**		**Number of function Evaluations**	
	Default	Nelder-Mead (improv. %)	PRO (improv. %)	Nelder-Mead	PRO (speedup)	Nelder-Mead	PRO
A1	4.385	3.473 (21%)	3.411 (22%)	41	11 (3.7)	41	188
A3	4.250	3.445 (19%)	3.288 (23%)	54	15 (3.6)	53	238
A4	5.656	4.831 (15%)	4.601 (19%)	55	13 (4.2)	55	139

Table 5.2 summarizes the results for all algorithms. PRO consistently finds better configurations for all classes. The percent improvement listed in the table for PRO and Nelder Mead shows the percentage improvement of the respective configurations against the default configuration. Speedup indicates how fast PRO finds configurations compared to the Nelder Mead algorithm - 3.7, 3.6 and 4.2 times faster for A_1, A_2 and A_3 respectively. Note that the number of PRO function evaluations is significantly larger than that for Nelder Mead. This additional expense is ameliorated by the parallel evaluation of candidate configurations. Moreover, for all our experiments, we use our database of previously evaluated candidate configurations to eliminate rerunning configurations previously measured. This also helps to reduce the overall tuning time.

5.2.2 High Performance Linpack (HPL)

HPL is a popular message-passing implementation of the Linpack benchmark. HPL solves an order N dense system of linear equations of the form $Ax=b$ using LU factorization. The matrix is divided into $NB \times NB$ blocks. The blocks are then dealt onto a $P \times Q$ processor grid a using block-cyclic data distribution. HPL is built on top of the Basic Linear Algebra Subroutine (BLAS) package. We used high-performance Goto BLAS [34] in our installation. The performance of the system is measured in GFlops/second. This measurement is provided as a part of the program output. The goal is to select a good matrix size N and blocking factor NB to maximize this metric. In addition, HPL exposes 15 other input parameters that can be set at launch time to tailor the execution of the code on different platforms. However, with very coarse-grained instructions on how to set these parameters, users are left with no choice but to hand-tune the parameters. Such hand-tuning is guided by semi-random guesses. Thus, finding a good input configuration is tedious and can take substantial time. We use Active Harmony's offline tuning mechanism to automate the search for input parameters. Once the parameter space definition is sent to the Harmony server, no intermediate feedback is required to guide the search process. With no default input configuration available, this experiment provides a strict comparison between PRO and Nelder Mead algorithms.

Table 5.3: Tunable input parameters for HPL

Parameter	**Domain**
$P \times Q$	Depends on the number of processors used (usually square grids are better)
N	Up to 80 % of the available memory, step size: 256
NB	32-256: step size: 2
pfact	left, right, crout
nbmin	2-10
ndiv	2-10
rfact	left, right, crout

We took note of previous research that studied the application behavior of HPL [79]. These results suggested that not all HPL parameters have noticeable impact on performance. Of course, parameters that do not affect performance in one system might have significant impact on another. We conducted a parameter study to determine what parameters had significant impact on performance on our cluster. We vary only one parameter at a time to try and measure its importance. After trying all the applicable options for a given parameter (when the other parameters are fixed), if the HPL performance remains roughly[1] the same for all options, we remove the parameter from our search space. After the study, we short-listed the parameters that have a noticeable impact on HPL performance on our system. This parameter list along with their domain values is provided in Table 5.3. Parameter *bcast* was set at *2rg* (increasing two-ring broadcasting method)[2] and *depth* at 0. *pfact* (and *rfact*) specifies panel (recursive) factorization method. *nbmin* specifies the number of sub-panels and *ndiv* specifies the number of columns in the recursive base case. The matrix size N should not exceed the amount of memory available across the nodes in the cluster. HPL guidelines suggest using 80% of available memory for the matrix to get the maximum performance leaving 20% for the Operating System and other background activities.

The experiment was conducted for 8 Nodes (16 CPUs). The performance metric (objective function) is R_{max}, which is the maximum measured HPL performance in GigaFlops/second. To calculate the efficiency of the system, we divide R_{max} by the theoretical peak performance, R_{peak}, for the system. R_{peak} is calculated by multiplying the total number of processors, the processor clock frequency and the theoretical number of 64-bit floating-point operations per clock.

Figure 5.3 shows the iteration history of both PRO and Nelder Mead algorithms. PRO input configurations reached 69.3% of R_{peak} after 11 iterations and the algorithm evaluated 96 unique candidate configurations in the process. Mean-

[1]We use a 3% threshold, i.e. if HPL performance numbers for different options (that a given parameter can take) remain within 3%, we eliminate the parameter from the search space. The 3% value was chosen because taking four performance samples for the same HPL run (with fixed parameters) achieves this threshold.

[2]Parameter *bcast* had no impact on the execution time of HPL on our Linux cluster. In fact, we can set this parameter to any other permissible value.

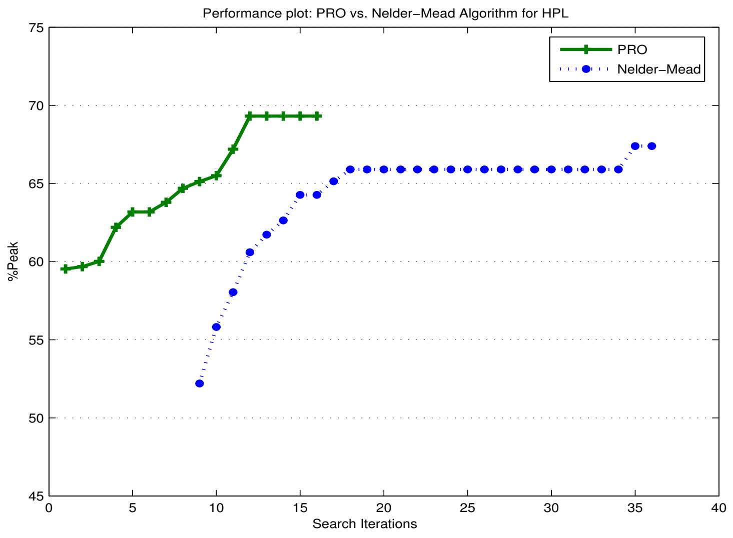


Figure 5.3: Comparison of best performing points of PRO and Nelder-Mead Algorithm

while Nelder-Mead configurations could not get more than 65.9% of R_{peak} for the first 33 iterations. The performance slightly improved to 67.4% after 33 iterations. In conclusion, PRO finds a better input configuration 3 times faster than Nelder-Mead simplex algorithm.

5.2.3 Parallel Ocean Program

The Parallel Ocean Program (POP) [28, 80] was developed at Los Alamos National Laboratory and is a descendant of Bryan-Cox-Semtner class of Ocean models first developed at the NOAA (National Oceanic and Atmospheric Administration) in Princeton, NJ in the late 1960s. Currently, POP is being used as the ocean component of the Community Climate System Model (CCSM). The program solves three-dimensional primitive equations for fluid motions on a sphere using hydrostatic and Boussinesq approximations. Spatial derivatives are computed using finite-difference discretizations which are formulated to handle any generalized orthogonal grid on a sphere, including dipole and tripole grids. POP provides approximately three dozen input parameters that can be changed at application launch time. With each parameter taking anywhere from 2 to 4 different values, the parameter space for POP is fairly large. After conducting a survey of parameters for their effect on the application performance on our platform, we narrowed down the list to 11 parameters, which are listed in Table 5.4. These parameters allow users to select among various numerical algorithms and physical parameterizations at launch time. For example, *solv_type* specifies which iterative method (pre-conditioned conjugate gradient, Jacobi or conjugate gradient residual) is used to solve a two-dimensional elliptical equation for the surface pressure.

Table 5.4: POP parameters

Parameter	**Domain**	**Default for *test***	**Default for *x1***	**PRO-value after 6 iterations**	**Nelder-Mead after 42 iterations**
solv_type	pcg, cgr, jac	pcg	pcg	cgr	cgr
tadvect_ctype	centered, upwind3	centered	upwind3	centered	centered
vmix_choice	const, kpp, rich	rich	rich	const	const
hmix_momentum_choice	del2, del4, anis	del2	anis	del2	del2
hmix_tracer_choice	del2, del4, gent	del2	gent	del2	del2
state_choice	jmcd, mwjf, poly, line	mwjf	jmcd	line	line
state_range_opt	ignore, check, enforce	enforce	ignore	ignore	enforce
ws_interp_type	nearest, linear, 4point	nearest	nearest	4point	4point
shf_interp_type	nearest, linear, 4point	nearest	nearest	linear	4point
sfwf_interp_type	nearest, linear, 4point	nearest	nearest	4point	4point
ap_interp_type	nearest, linear, 4point	nearest	nearest	4point	4point

For our experiments, we use the following two benchmarks: *test* and *x1*. The first benchmark comes bundled with the POP distribution and is used for validation and performance tuning. The model grid (192×128×20) generated internally is an equally-spaced latitude-longitude global grid with idealized land-masses. The *x1* benchmark is set up to be identical to the actual production configuration of the Community Climate System Model [41]. The model grid (320×384×40), topography, initial state, equation of state coefficients and other benchmark specifications for *x1* are available at the POP website [70]. Default parameter values for both the test and the *x1* benchmarks are provided in Table 5.4. In our experiment, tuning is done using the *test* benchmark. The benchmark is run for 20 timesteps on 32 processors. Input configurations generated by PRO and the Nelder Mead algorithm are then used to measure the execution time of *x1* (which is run on 32 processors for 20 simulation days).

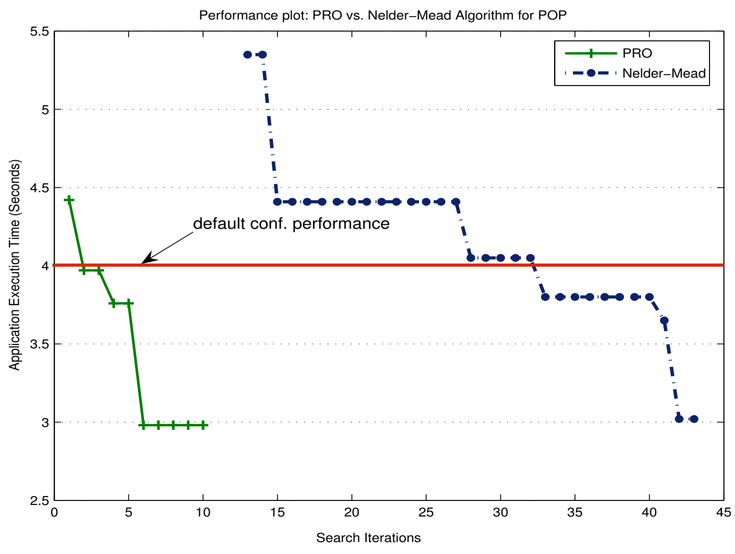


Figure 5.4: Comparison of the best performing points of PRO and Nelder-Mead Algorithm

Figure 5.4 plots the best points of both algorithms at different iterations. The initial 12 Nelder Mead iterations are not shown in the graph. PRO achieves a 26.4% improvement in execution time (for the test benchmark) in just 6 iterations and evaluates 52 unique input configurations in the process. PRO converges to a point in the 11th iteration. Nelder-Mead also out-performed the default configuration by a similar margin. However, it took over 7 times as many iterations. Table 5.4 shows the best configuration found by PRO after 6 iterations and that found by Nelder Mead after 42 iterations.

Table 5.5 summarizes the results for this experiment. Both the PRO and Nelder Mead input configurations perform well on the production sized runs of *x1* and reduce the execution time by 58%. The data from actual production sized runs for POP allows us to explore the relationship between the tuning time and the execution time of production size simulations. The tuning time for 6 PRO iterations is approximately 125 seconds (which is the sum of the worst performing measurements at each iteration) and the tuning time for 42 Nelder Mead iterations is approximately 840 seconds. Only one production run of *x1* amortizes the cost of tuning using Active Harmony and PRO. Generally, production runs consist of multiple runs as part of a parameter sweep, so in practice Harmony would provide significant gains in execution time.

Table 5.5: Summary of results for POP

Algo	**Execution Time (seconds)**			**Number of Iterations**		**Tuning Time (seconds)**	
	Default	Nelder-Mead (improv. %)	PRO (improv. %)	Nelder-Mead	PRO (speedup)	Nelder-Mead	PRO (speedup)
test	4.02	2.98 (26%)	2.96 (26%)	42	6 (7)	840	125 (6.7)
x1	1,246.61	510.9 (59%)	513.1 (59%)	-	-	-	-

5.3 Summary

In this chapter, we used Active Harmony (in offline auto-tuning mode) to automatically select appropriate input configurations for parallel programs. We evaluated PRO within the Active Harmony system and studied the performance of the algorithm on three well-studied benchmark codes: PSTSWM, HPL, and POP. We compared the performance of PRO with the Nelder Mead Simplex algorithm and show that PRO finds better input configurations up to 7 times faster. We showed that tuning for input parameters can reduce the application execution time significantly. Using PRO to tune input parameters, we reduced the execution time of PSTSWM by 23%. For HPL, an input configuration that achieved 69.3% of R_{peak} for 8-Nodes (16 CPUs) was discovered in just 11 iterations. For POP, we were able to increase performance by over 26% in just 6 PRO iterations. A production sized run of POP showed a 59% improvement in execution time by using the configuration found by our search algorithm.

Chapter 6

Compiler based auto-tuning

In this chapter, we describe a scalable and general-purpose framework for auto-tuning compiler-generated code. We combine Active Harmony's parallel search backend with the CHiLL compiler transformation framework to generate in parallel a set of alternative implementations of computation kernels and automatically select the one with the best-performing implementation. Our framework provides a general-purpose and scalable solution to code optimization with minimum or no feedback from the users. The resulting system achieves performance of compiler-generated code comparable to the fully automated version of the ATLAS library for the tested kernels. We start the chapter by providing a high-level motivation for a general purpose auto-tuner for compiler-generated search spaces.

6.1 Motivation

Today's complex architecture features and deep memory hierarchies require applying nontrivial optimization strategies on loop nests to achieve high performance. This is even true for a simple and well-studied loop nest like Matrix Multiply. Although naively tiling all three loops of Matrix Multiply would significantly increase its performance, the performance is still well below hand-tuned libraries. Chen et al [16] demonstrate that automatically-generated optimized code can achieve performance comparable to hand-tuned libraries by using a more complex tiling strategy combined with other optimizations such as data copy and unroll-and-jam. Combining optimizations, however, is not an easy task because loop transformation strategies interact with each other in complex ways.

Different loop optimizations usually have different goals, and when combined they might have unexpected (and sometimes undesirable) effects on each other. Even optimizations with similar goals but targeting different resources, such as unroll-and-jam plus scalar replacement targeting data reuse in registers, and loop tiling plus data copy for reuse in caches, must be carefully combined. The unroll factors must be tuned so that reuse in registers is exploited without causing register spilling or instruction cache misses. On the other hand, tiling plus data copying for reuse in caches changes the iteration order and data layout, and may affect reuse in registers. When combining unroll-and-jam and tiling, both unroll and tile sizes must be tuned so that performance gains are complementary. To illustrate this point, consider again Figure 1.1. This graph illustrates these complex interactions by showing the performance of square matrix multiplication as a function of tiling and unrolling factors[1]. Tiling factors range from 2 to 80 and unrolling factors from 2 to 32. We see a corridor of best performing combinations along the x-y diagonal where tiling and unrolling factors are equal, and smaller corridors when tile factors are multiples of unroll factors. The best performing code variant used a tiling factor of 24 and unrolling factor of 24 and achieves a performance of 845 MFLOPS.

[1]Recall from chapter 1 that we took a naive Matrix Multiplication implementation. All three loops were tiled with the same tile-size. Only the innermost loop was unrolled.

Empirical optimization can compensate for the lack of precise analytical models by performing a systematic *search* over a collection of automatically generated code variants. Each variant exposes a set of parameters that controls the application of different transformation strategies. Parameter configurations for variants serve as points in the search space and the objective function values[2] associated with the points are gathered by actually running the variants on the target architecture. The success of empirical search is largely driven by how well the chosen *search* algorithm navigates the search space. The search space shown in Figure 1.1 is not smooth and contains multiple minimas and maximas. The best and the worst configurations are a factor of six different.

Finding a good set of loop transformation parameters is an example of the type of search that the Active Harmony system is designed to address. Our system provides a selection of search algorithms designed specifically to deal with search spaces where the explicit definition of the objective function is not available. In the next section, we describe a specific modification that we made to the original PRO algorithm to make it suitable for searching compiler generated parameter spaces.

6.2 Parallelizing Expansion Check Step

Recall that each simplex transformation step generates up to $kN-1$ new vertices. The time required to complete the parallel evaluation of these new vertices is the time taken by the worst performing vertex. Recall that the decision to in-

[2]The objective function values associated with points in the search space can be any desired metric of performance (for example - time per timestep, MFLOPS, cache utilization etc.).

troduce the expansion-check step in PRO was motivated by the observation that there are some expansion points with very poor performance. For online tuning of SPMD-based parallel applications, such configurations slow down not only the search but also the execution of the application itself. To avoid these time consuming instances, before evaluating all expansion points, PRO first calculates the expansion point performance of only the most promising case at the expense of parallelism. If the expansion checking step is successful, the algorithm performs expansion of other points in the simplex.

In an offline parallel search, however, processors participating in the search are independent and do not have to finish evaluating the configuration that they were assigned. This allows us to take full advantage of the underlying parallelism while still avoiding expansion points with poor performance. To that end, the modified PRO (henceforth referenced as PRO-C) evaluates all expansion points and the decision to accept or reject the expanded simplex is based on the performance of the most promising case. If the performance reported by the most promising case is worse than that of the best point in the reflected simplex, our system sends a signal to all the other processors to stop the evaluation of their candidate configurations and accepts the reflected simplex. The expansion of the simplex is accepted if the performance of the most promising case is better than the best vertex in the reflected simplex. With this modification, we not only reduce the number of steps within one iteration of the search algorithm to at most two (reflection-expansion and reflection-shrink) but also increase parallelism.

6.3 System Design

In this section, we describe our system design. We divide the discussion into two parts. First, we provide a brief and high-level overview of CHiLL — a loop transformation and code generation framework that is capable of generating code-variants based on the parameter values supplied by the Active Harmony server. Second, we describe how CHiLL and Active Harmony interact with each other to generate a set of alternative implementations of computation kernels and select the best-performing implementation.

6.3.1 Loop Transformation Framework: CHiLL

CHiLL [14, 15] is a polyhedral loop transformation and code generation framework. CHiLL's high-level script interface allows compilers or application programmers to use a common interface to describe parameterized code transformations to be applied to a computation, whose parameters can be instantiated by an external search engine. In CHiLL nomenclature, these scripts are called "recipes" (we provide example recipes later in this chapter). Besides making it easy to interface with the code-generation utility, these code transformation recipes offer an additional advantage. *Unlike traditional compiler optimizations which must be coded into the compiler, these recipes can be evolved and reused over time.* A recipe library, created by compiler experts and developers based on their experience working with real codes, can then be consulted by auto-tuners to tune arbitrary loop-nests.

6.3.2 Overall System Workflow

Figure 6.1 shows the overall workflow of our system. The code transformation recipes and parameter specifications (i.e. parameter domain and constraints) can be either generated by the compiler automatically or by the users tuning their application code. With this flexibility, our approach can support both fully automated compiler optimizations and user-directed tuning. For our experiments, we translate loop transformation sequences from the algorithms presented by Chen et al [16] to CHiLL scripts. Specifications for unbound parameters in the scripts are derived using simple heuristics based on architectural parameters (e.g., by considering cache capacity to generate constraints for tile-sizes). We elaborate more on parameter specification in the next section. If a user, with domain knowledge, wants more control over what part of the parameter space to focus on, he/she can provide additional constraints to fine-tune the search space. Using the parameter specifications, we normalize the domain of each parameter onto our internal integer based coordinate system. This step is necessary to ensure that the differences in the range of values parameters can take in different dimensions do not unduly influence the L_1 distance metric.

Parameters that appear in one or more constraints are considered to be interdependent and are evaluated as sets. For example, tile-size parameters for multiple loops may appear in one or more cache capacity constraints. A simple constraint solver is then used to enumerate points for each of these sets. Projection of an inadmissible point to a valid point in the search space is done (by the projection

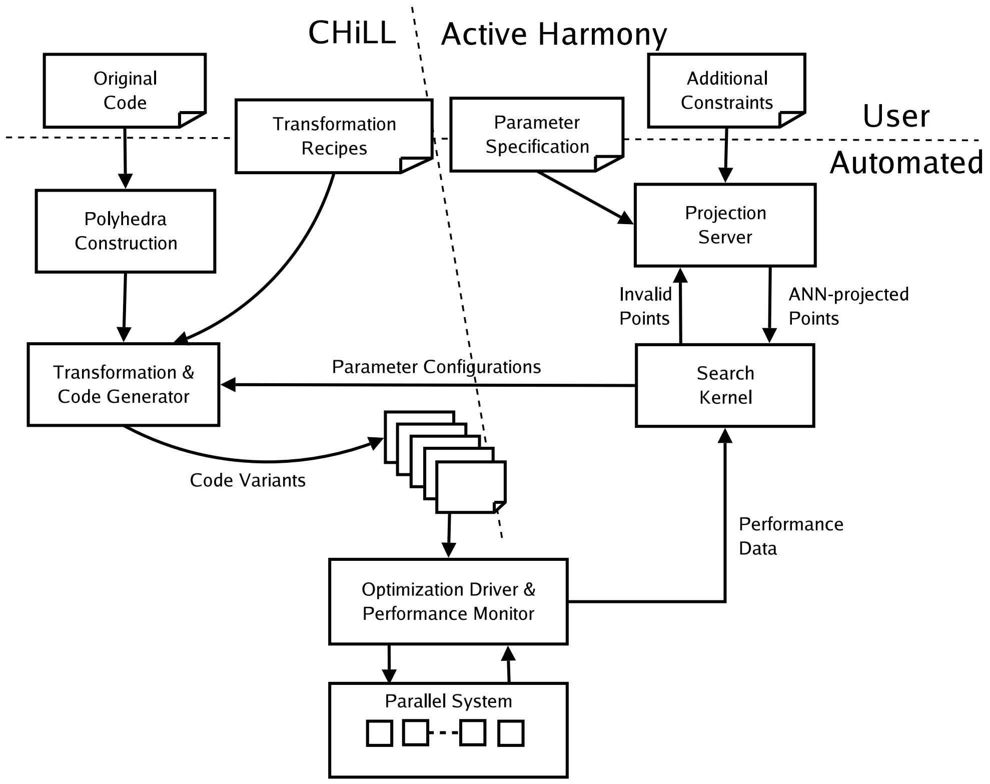


Figure 6.1: Overall system workflow diagram

server) separately for different groups of parameters.

At each search step, Active Harmony first generates a set of parameter configurations. These configurations are evaluated by the projection server. The projection server projects the inadmissible points (if any) to nearest admissible points. Active Harmony server then requests CHiLL to generate code variants with given sets of parameters for loop transformations. The optimization driver waits until the code-generation process is complete. The CHiLL generated code variants are then compiled and evaluated in parallel on the target architecture by the optimization driver. Measured performance values are consumed by the search-kernel to make simplex transformation decisions.

Table 6.1: Compiler-based tuning: top table shows kernels used for experiments. Bottom table provides the transformation recipes and constraints

Kernel	*Naive Code*
MM	`DO K = 1, N` `DO J = 1, N` `DO I = 1, N` `C[I,J] = C[I,J]+A[I,K]*B[K,J]`
TRSM	`DO J = 1, N` `DO K = 1, N` `DO I = K + 1,N` `B(I,J) = B(I,J) - B(K,J)*A(I,K)`
Jacobi	`DO K = 2, N-1` `DO J = 2, N-1` `DO I = 2, N-1` `A(I,J,K) = C*(B(I-1,J,K)+B(I+1,J,K)+` `B(I,J-1,K)+B(I,J+1,K)+` `B(I,J,K-1)+B(I,J,K+1))`

Kernel	*Transformation Recipe*	*Constraints*
MM	`permute([3,1,2])` `tile(0,2,TJ)` `tile(0,2,TI)` `tile(0,5,TK)` `datacopy(0,3,2,1)` `datacopy(0,4,3)` `unroll(0,4,UI)` `unroll(0,5,UJ)`	$TK \times TI \leq \frac{1}{2}\left(\frac{size_{L2}}{2}\right)$ $TK \times TJ \leq \frac{1}{2}\left(\frac{size_{L1}}{2}\right)$ $UI \times UJ \leq size_R$ $TI,\, TJ,\, TK \in [0, 2, 4, \ldots, 512]$ $UI,\, UJ \in [1, 2, \ldots, 16]$
TRSM	`permute([1,3,2])` `tile(0,3,TK)` `split(0,2,L3>=L1+TK)` `tile(0,3,TI,2)` `tile(0,3,TJ,2)` `datacopy(0,3,2)` `datacopy(0,4,3,1)` `unroll(0,4,UJ1)` `unroll(0,5,UI1)` `datacopy(1,2,3,1)` `unroll(1,2,UJ2)` `unroll(1,3,UI2)`	$TK \times TK \leq \frac{1}{2}\left(\frac{size_{L2}}{2}\right)$ $TK \times TJ \leq \frac{1}{2}\left(\frac{size_{L1}}{2}\right)$ $TK \times TI \leq \frac{1}{2}\left(\frac{size_{L2}}{2}\right)$ $UI1 \times UJ1 \leq size_R$ $UI2 \times UJ2 \leq size_R$ $TI, TJ, TK \in [0, 2, 4, \ldots, 512]$ $UI1, UJ1, UI2, UJ2 \in [1, 2, \ldots, 16]$
Jacobi	`original()` `tile(0, 3, TI)` `tile(0, 3, TJ)` `tile(0, 3, TK)` `unroll(0,5,UJ)`	$TI, TJ, TK \in [0, 2, 4, \ldots, 512]$ $UJ \in [1, 2, \ldots, 16]$

6.4 Empirical Results

In this section, we present an experimental evaluation of our offline auto-tuner. First, we use a Matrix Multiplication kernel to explore the effectiveness of

PRO-C (the modified PRO) on the search space for loop transformation parameters. We study how the size of the initial simplex (and hence the degree of parallelism) affects the convergence and performance of the search algorithm. In the second part, we use our framework to optimize two additional computational kernels — Triangular Solver (TRSM) and Jacobi. The use of linear algebra kernels — Matrix Multiplication and Triangular Solver - was motivated by our goal to compare the effectiveness of our framework to well tuned codes. The results for the Jacobi kernel show that our framework is general-purpose and that it can handle arbitrary code beyond the linear algebra library domain. For all the kernels, we provide the original code, the transformation recipe and the constraints on unbound parameters in Table 6.1.

The experiments were performed on the umd-cluster (see section 5.2). We compare the performance of our code versions with those of the native compiler (ifort 10.0.026, compiled with -O3 -xN). When compiling our transformed code, we turn off the native compiler's loop transformations to prevent the compiler from interfering with our optimizations. For Matrix Multiplication and Triangular Solver, we present the performance of ATLAS (version 3.8) self-tuning libraries. In addition to a near exhaustive sampling of the search space, ATLAS uses carefully hand-tuned BLAS routines contributed by expert programmers. To make a meaningful comparison, we provide the performance of the *search-only* version of ATLAS - code generated by the ATLAS Code Generator via pure empirical search. The search-only version was generated by disabling the use of architectural defaults and turning off the use of hand-coded BLAS routines.

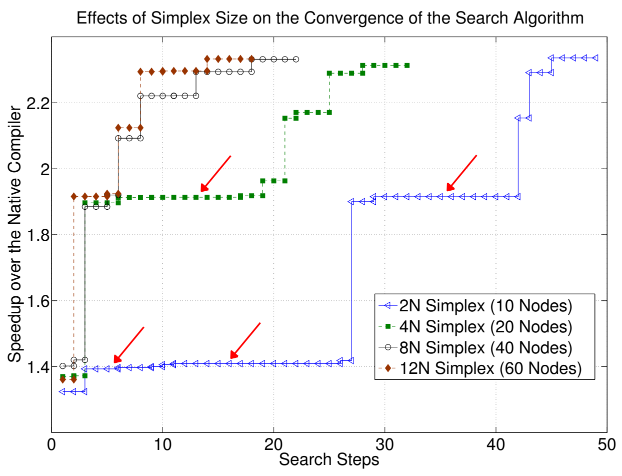


Figure 6.2: Effects of different degree of parallelism on the convergence of PRO-C

For all our experiments, unroll factors and tile sizes are constrained by the storage capacity of their associated memory hierarchy levels. In addition, for tile sizes, we use a simple heuristic which tries to fit references with temporal reuse into half of the cache, leaving the other half for references with spatial or no reuse.

6.4.1 Performance of PRO-C

In this section, we use Matrix Multiplication (MM) to demonstrate the effectiveness of parallel search. The optimization strategy reflected in the transformation recipe in Table 6.1 exploits the reuse of $C(I, J)$ in registers, and the reuse of $A(I, K)$ and $B(K, J)$ in caches (A and B have the same amount of temporal reuse, carried by different loops). The transformation recipe applies tiling to B in the L1 cache and A in the L2 cache. Data copying is applied to avoid conflict misses. In addition, to expose SSE optimization opportunities to the Intel compiler, the copying of A transposes the data into the temporary array. The values for the five unbound parameters TI, TJ, TK, UI and UJ are determined by the search algorithm.

Table 6.2: MM results - alternate simplex sizes

	$2N$	$4N$	$8N$	$12N$
Number of Function Evals.	276	571	750	961
Number of Search Steps	49	32	22	18
Speedup over Native	2.30	2.33	2.32	2.33

To study the effect of simplex size, we considered four alternative simplex sizes - $2N$ (10 Nodes), $4N$ (20 Nodes), $8N$ (40 Nodes) and $12N$ (60 Nodes), where N is the number of unbound parameters ($N = 5$ for this experiment). Each simplex was constructed around the same initial point, which was randomly selected from the search space at the beginning of the experiment. The search algorithm was run for a square matrix of size 800×800. The results for this experiment are summarized in Table 6.2.

Figure 6.2 shows the performance of the best point in the simplex across search steps. Search conducted with $12N$ and $8N$ simplices clearly use fewer search steps than the search conducted with smaller simplices. Recall from our discussion in section 6.1 and from Figure 1.1 that loop transformation parameter space is not smooth and contains multiple local minimas and maximas. The existence of long stretches of consecutive search steps with minimal or no performance improvement (marked by arrows in Figure 6.2) in $2N$ and $4N$ cases show that more search steps are required to get out of local minimas for smaller simplices. At the same time, by effectively harnessing the underlying parallelism, $8N$ and $12N$ simplices evaluate more unique parameter configurations (see Table 6.2) and get out of local minimas at a faster rate.

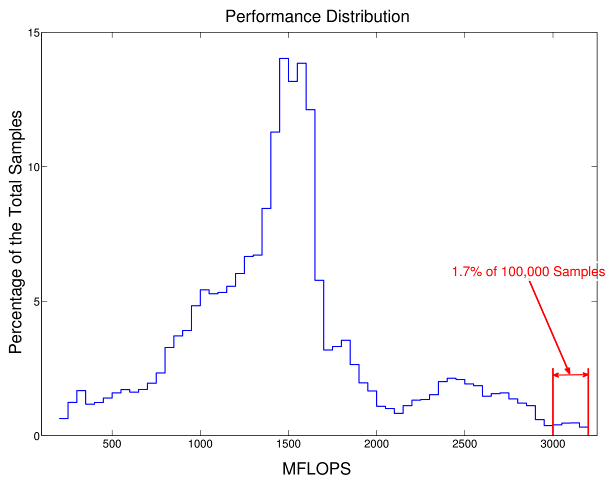


Figure 6.3: Performance distribution for randomly chosen MM configurations

The results summarized in Table 6.2 also show that as the simplex size increases, the number of search steps decreases, thereby confirming the effectiveness of increased parallelism. Using a $12N$ initial simplex, the search converges to a solution 2.7 times faster than using $2N$ initial simplex.

The next question regarding the effectiveness of our framework relates to the quality of the search result. To answer this question, we selected 100,000 uniformly distributed samples from the search space, which has over 70 million total points, and evaluated the performance associated with all the samples. The performance distribution is shown is Figure 6.3. Approximately 1.7% of the total samples report performance greater than 3 GFLOPS. The best performance (3.22 GFLOPS) was associated with the configuration $TI = 160$, $TJ = 6$, $TK = 162$, $UI = 1$ and $UJ = 6$. For the same problem size, our code delivers 3.17 GFLOPS. The result demonstrates PRO-C's effectiveness on compiler-generated search spaces.

Finally, figure 6.4 shows the performance of the code variant produced by a

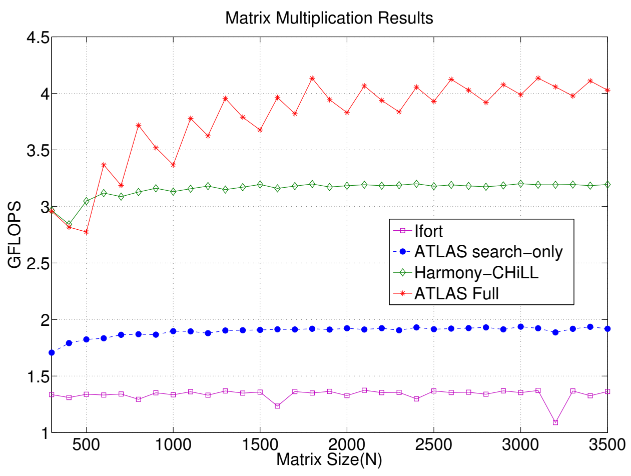


Figure 6.4: Results for MM kernel

$12N$ simplex across a range of problem sizes along with the performance of native compiler, ATLAS' search-only and full version. Our code version performs, on average, 2.36 times faster than the native compiler. The performance is 1.66 times faster than the search-only version of ATLAS. Our code variant also performs within 20% of ATLAS' full version (with processor-specific hand coded assembly).

6.4.2 Triangular Solver (TRSM)

The optimization strategy for the TRSM kernel is outlined in its transformation recipe provided in Table 6.1. Two inner loops are permuted to reuse $B(I, J)$ in registers, and loops I and J are unrolled. For data reuse in cache, loop K is tiled first. The splitting condition is based on the decision to separate read access $B(I, J)$ from write access $B(K, J)$. After splitting, one subloop has non-overlapping read and write accesses and it is optimized in the same way as matrix multiplication. The other subloop has only one non-overlapping read access $A(I, K)$, for which data copy is applied to reduce cache conflict misses caused by this array reference.

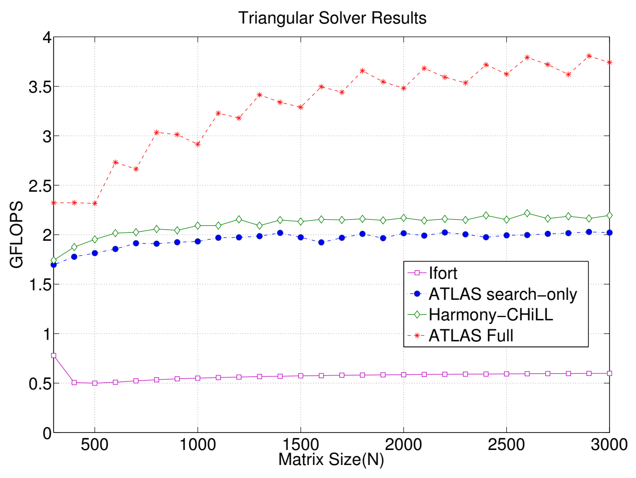


Figure 6.5: Results for TRSM kernel

Unbound parameters in the transformation recipe TI, TJ, TK, $UI1$, $UJ1$, $UI2$ and $UJ2$ form a seven dimensional parameter space. PRO-C used a 60-point simplex and converged to a solution in 55 steps evaluating 1,579 unique parameter configurations. Figure 6.5 shows the performance of the code variant along with the performance of the Native compiler and both ATLAS versions. The parameter configuration selected by PRO-C performs, on average, 3.62 times faster than the native Intel compiler. The performance, on average, is 1.07 times faster than the search-only version of ATLAS. However, ATLAS full-version (with processor-specific hand-tuned assembly) performance is 1.55 times faster than our code-variant.

6.4.3 Jacobi

The transformation recipe provided in Table 6.1 outlines the optimization strategy we use for this kernel. Since only array B has reuse on three dimensions, the loops are tiled on three dimensions for reuse in L1 or L2 cache. Arrays A and B access data in the loop nest in the same order as the dimensionality of the iteration

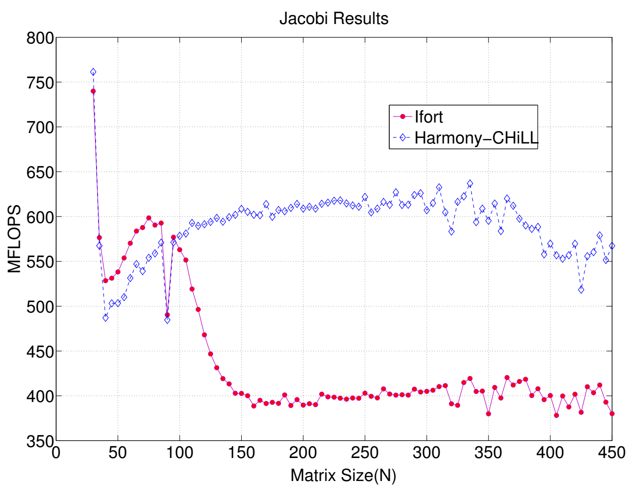


Figure 6.6: Results for Jacobi kernel

space, thus the original loop order is best for spatial reuse in cache and TLB. Finally loop J is unrolled for register reuse. Four unbound parameters in the script TI, TJ, TK and UI form a four-dimensional parameter space.

PRO-C took 23 steps (870 unique function evaluations) to converge to $TI = 0$, $TJ = 22$, $TK = 0$ and $UJ = 1$. The results of $TK = 0$ and $TI = 0$ suggest that no tiling is needed for K and I loops. Tiling only the J loop produces the best performance. Also no unroll is performed. We suspect that the native compiler's scalar replacement cannot take advantage of available register reuse across the I dimension so there is little benefit of unrolling J. Figure 6.6 shows the performance of our code variant. Note that we did not include a comparison to ATLAS for Jacobi because it is not a linear algebra kernel and therefore there is no ATLAS implementation available. On average, our code variant performs 1.35 times faster than the native Intel compiler.

6.5 Summary

In this chapter, we described a scalable and general-purpose framework for auto-tuning compiler-generated code. We showed for three benchmark kernels that with automatic compilation and tuning in parallel, we can achieve performance that greatly exceeds that of the native compiler, and is comparable to near-exhaustive search of the ATLAS library system. Performance for various kernels is 1.4 to 3.6 times faster than the Native Intel compiler without search.

We note that we are not trying to compete with linear algebra library generators such as ATLAS. Our goal is to provide a general-purpose compiler based framework, which can generate and evaluate different optimizations that can be applied on arbitrary application codes. In the absence of a general-purpose framework, manual exploration of possible optimizations can be prohibitively time consuming and painful for a programmer.

Chapter 7

Whole Program Tuning

In chapter 6 we showed that for well-defined benchmark kernels (such as matrix multiplication), compiler-based offline auto-tuning can deliver significant improvements over the optimizations offered by native compilers. That success sparked an interest in extending the search-based empirical auto-tuning methodology to arbitrary program components and whole programs. Shifting the focus from empirically tuning a few kernels to tuning whole programs will certainly help avoid the enormous productivity costs associated with tuning and retargeting applications to next generation exascale systems. However, the shift also comes with its own set of challenges. The first challenge stems from the fact that compute intensive loop-nests in full applications are often wedged in the middle of large monolithic code sections. Code outlining tools are needed to extract these loop-nests to separate standalone functions. These outlined codes can be more easily managed, analyzed and transformed by loop-transformation tools. In addition, code-outlining process helps reduce the challenging whole program tuning problem into a set of manageable kernel tuning tasks. We use ROSE compiler framework to do the code-outlining [54]. The second challenge is related to the number of code-variants for a complete application. This number can be fairly large. Therefore, strategies to judiciously select what transformation techniques to apply to different sections of the application code are needed to keep the tuning time at manageable levels. We work directly with compiler experts and application developers to make these decisions. Furthermore, compiler-based auto-tuning requires a code-transformation framework that is able to generate different codes rapidly during the search by adjusting parameter values. It also demands that the compiler have a clean interface to a separate parameter search engine. We use CHiLL (described in section 6.3.1), which provides a high-level script interface to describe code transformation sequences, to transform ROSE-outlined functions.

The auto-tuning framework presented in this chapter combines Active Harmony with ROSE's outliner and CHiLL compiler transformation framework. We use a real application benchmark, SMG2000 [10], as a subject application. The auto-tuning process is driven by Active Harmony which utilizes the outlined code and the code-transformer to search for the best performing variants of outlined loop-nests. In the next section, we describe the overall tuning workflow.

7.1 Overall Workflow

Figure 7.1 shows the overall workflow of our system. The tuning process starts by first using application profiling tools (such as HPCToolkit [2]) to identify computationally intensive loop-nests (not shown in the figure). The ROSE outliner outlines the kernels to separate and independently compilable C source files with all dependent structures and typedef declarations preserved. Code-outlining is a one-time process — outlined kernels can be reused in subsequent auto-tuning runs.

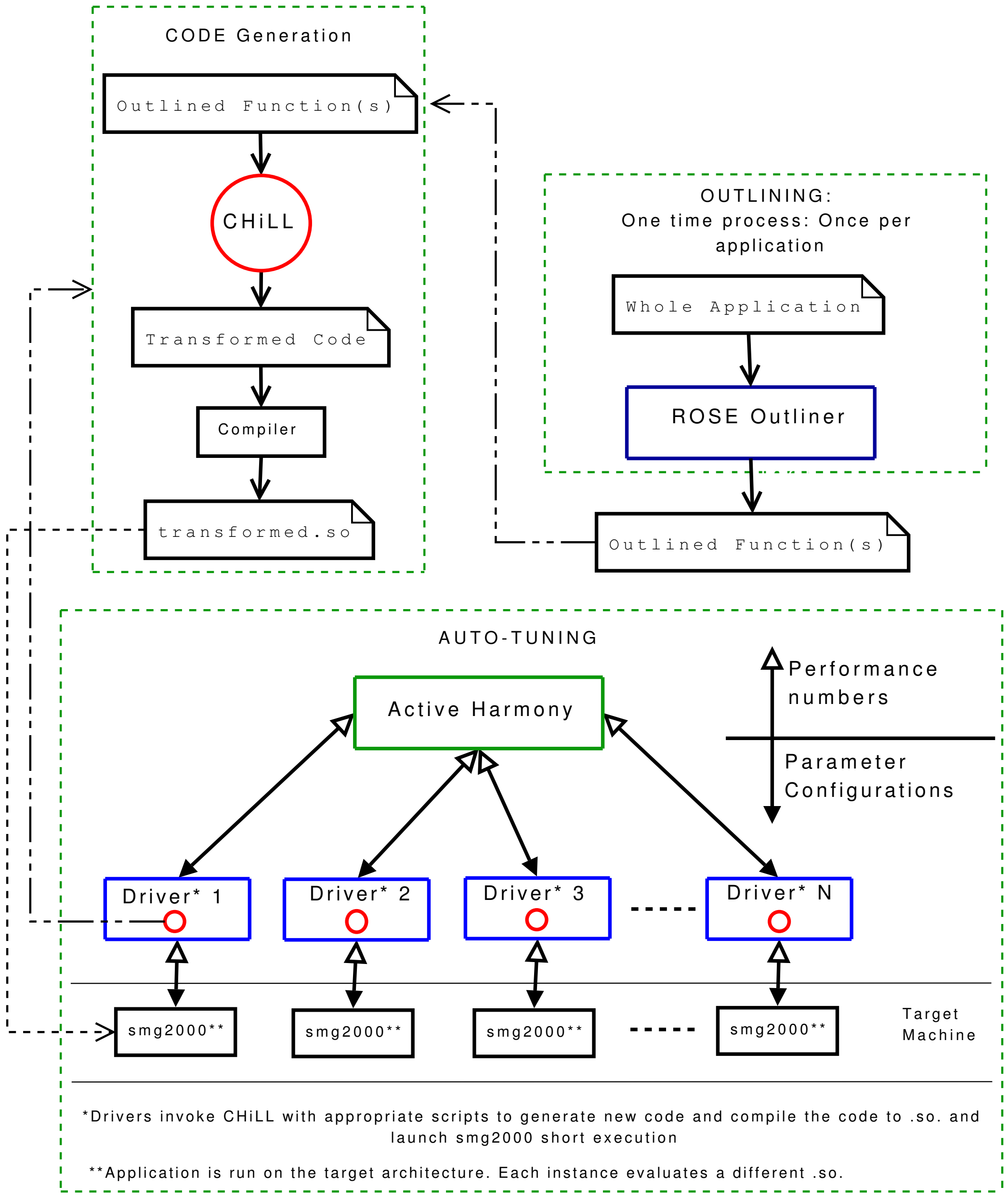


Figure 7.1: Overall workflow: SMG2000 tuning

Application developers make simple modifications to the driver code that we provide as a part of the software release package. These changes are made to export application-specific tuning options to the Active Harmony server. The driver, which can be run on the login nodes of a parallel machine, connects to a given Active Harmony server and requests candidate parameter configurations. The driver then invokes CHiLL to generate variants of the outlined kernel based on the code transformation parameters supplied by the Active Harmony server. The code generated on-demand is compiled into a shared library. Once the new code is ready, the application is run on the target machine. The application dynamically loads the transformed kernel by using the dlopen/dlsym mechanism. Once the execution is complete, the driver collects performance measurement and sends them to the Active Harmony server. The process continues for a specified number of iterations or until the search algorithm converges to a point in the search space. For parallel search, we run multiple copies of the driver. The number of copies is determined by the number of tunable parameters and the simplex size (which is, in turn, determined by the available resources). The use of shared library mechanism helps to keep the tuning time short because only the outlined and transformed code has to be recompiled between successive tuning runs.

7.2 Subject Application: SMG2000

We consider the SMG2000 [10] benchmark as a subject application. SMG2000 is a parallel semi-coarsening multigrid solver for the linear systems arising from finite difference, finite volume, or finite element discretizations of the diffusion equation on logically rectangular grids (equation 7.1).

$$\bigtriangledown \cdot (D \bigtriangledown u) + \sigma u = f \tag{7.1}$$

The code solves both 2D and 3D problems with discretization stencils of up to nine points in 2D and up to twenty seven points in 3D. The most time-consuming kernel (approximately 55% of the execution time on the target system used for the experiments) in the SMG2000 benchmark is shown in Table 7.1. The kernel consists of sparse matrix vector multiplication expressed in four-deep loop-nest[1]. The kernel performs a stencil computation by sweeping the same array data (accessed using the inner `i`, `j`, and `k` indices) multiple times for each stencil element (the outermost `s` index). Thus, the kernel lacks data reuse and causes excessive cache misses [54].

To minimize the time required for tuning, many offline auto-tuners use "representative short application executions". In this technique, the application being tuned is run with a meaningful input data for a short period of time and tuning modifications are made between successive short executions [19]. Recall that the objective function values associated with different parameter configurations are derived by running the application on the target machine. Therefore, representative short runs help reduce the overall time required for offline auto-tuning. SMG2000 execution is divided into three distinct phases — `initialization`, `setup` and `solve`. All three phases make several calls to the outlined function. We disable the `solve` phase and record the total time spent in the outlined kernel during the `initialization` and `setup` phases.

7.3 Empirical Results

The auto-tuning experiments were performed on a 64-node Linux cluster. Each node is equipped with dual-core Intel Xeon 2.66 GHz (SSE2) processor. L1- and L2-

[1]The outlined kernel shown is a simplified version. Actual code is less clean.

Table 7.1: SMG2000 tuning: top table shows the kernel. Bottom table provides the transformation recipe and constraints (SMG2000)

$Kernel$	$Original$ $Code$

$Matrix - vectorMultiply$	`for (si = 0; si < stencil_size; si++)`
	` for (kk = 0; kk < hypre__mz; kk++)`
	`  for (jj = 0; jj < hypre__my; jj++)`
	`   for (ii = 0; ii < hypre__mx; ii++)`
	`    rp[((ri+ii)+(jj*hypre__sy3))+(kk*hypre__sz3)] -=`
	`     ((Ap_0[((ii+(jj*hypre__sy1))+`
	`      (kk*hypre__sz1))+(((A->data_indices)[i])[si])])*`
	`      (xp_0[((ii+(jj*hypre__sy2))+(kk*hypre__sz2))+`
	`      (( *dxp_s)[si])]));`

$Kernel$	$Transformation$ $Recipe$	$Constraints$

$Matrix - vectorMultiply$	`permute([2,3,1,4])`
	`tile(0,4,TI)`
	`tile(0,3,TJ)`
	`tile(0,3,TK)`
	`unroll(0,6,US)`
	`unroll(0,7,UI)`
		$0 \leq TI \leq 122$
		$0 \leq TJ \leq 122$
		$0 \leq TK \leq 122$
		$0 \leq UI \leq 16$
		$0 \leq US \leq 10$
		$compilers \in \{gcc, icc\}$

cache sizes are 128 KB and 4096 KB respectively. Active Harmony uses the Parallel Rank Ordering (PRO) algorithm to navigate the search space. Short executions of SMG2000 are done in parallel on the target machine, with each execution instance using a different code-variant. Transformation parameters are adjusted and corresponding new code-variants are generated between successive runs of SMG2000. The search uses a 24-point simplex, which means up to 23 new code-variants are evaluated in parallel at each search-step.

The optimization strategy (expressed in terms of a CHiLL-recipe) and constraints on transformation parameters are provided in Table7.1. The recipe tiles the `i`, `j` and `k` loops (with `TI`, `TJ` and `TK` tiling factors) to improve data reuse in caches. The stencil loop and the innermost loop are unrolled (with `US` and `UI` unrolling factors) to improve reuse in registers. The search-space is six-dimensional and includes

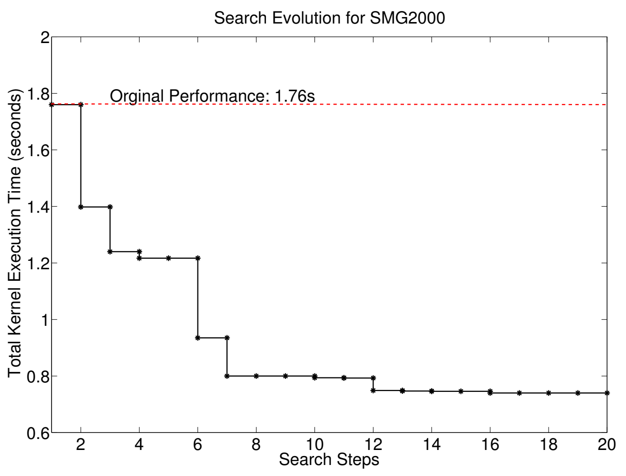


Figure 7.2: Search evolution for offline SMG2000 tuning

a parameter that chooses between two compilers to compile the transformed kernel — `gcc` and `icc`.

The search converges in 20 steps. The search-evolution (performance of the best-point at each search-step) is shown in Figure 7.2. The y-axis shows the total time spent in the outlined kernel(in seconds) per short representative SMG2000 execution. The x-axis shows the PRO search steps. The configuration that PRO converges to is: `TI=122`, `TJ=106`, `TK=56`, `UI=8`, `US=3`, `comp=gcc`[2]. The performance improvement is $2.37X$ of the time for the outlined kernel. We then use the code-variant associated with this parameter configuration to do a full run of SMG2000 (with input parameters `-n 120 120 120 -d 3`). The results from full SMG2000 run are summarized in Table 7.2. Full application execution improves by 27.2%.

[2]This was gcc version 4.1.2 and icc version 10.0.026, where icc has been known to have poor performance.

Table 7.2: SMG2000: full run performance

Auto-tuned	Original	Improvement
$49.86s$	$68.52s$	27.2%

7.4 Summary

In this chapter, we combined ROSE's code-outliner, CHiLL loop-transformer and Active Harmony to create a general-purpose offline auto-tuner that can handle arbitrary program components and full applications. We demonstrated the benefits of the system on a real scientific application benchmark — SMG2000. We showed how these three components complement each other and work together to create an integrated framework that supports code-outlining, automatic compilation and parallel search and pinpoints a code-variant that perform 2.37 times faster than the original loop nest. When the full application is run using the code variant found by the system, the application's performance improves by 27%.

Chapter 8

Runtime auto-tuning

In this chapter, we extend on the work presented in chapter 7 and present a runtime compilation and tuning infrastructure designed to improve the performance of parallel applications within a single execution. A unique feature that distinguishes the work presented here from other runtime auto-tuners is that our system can also handle runtime tuning for tunable parameters that require code generation (for example, different unroll factors). For such parameters, our auto-tuner generates and compiles new code on-the-fly. Effectively, *we merge traditional feedback directed optimization and just-in-time compilation.* We show that our system can leverage available parallelism in today's HPC platforms by evaluating different code-variants on different nodes simultaneously.

Based on the input dataset, a given parallel application can have vastly different execution profiles. Input datasets can specify physical domain, solver type(s), solver parameters, discretization order, and so on. Taking an offline tuning approach to tune for all possible computational bottlenecks is not a tractable goal. Instead, on-demand tuning during production execution is a desirable approach. This on-demand approach also benefits from the availability of real-time performance data, which can be linked back to specific code sections and architecture-specific features. This information is generally not available at compile time and even when some information is available, most compilers choose not to use the information. The compilers are designed to be generic and as such, they base their optimization decisions on simple and conservative analytical models. With continuous dynamic tuning, more aggressive tuning strategies can be explored. Furthermore, some tuning decisions made by the compiler can be undone at runtime should they interact with more profitable optimizations negatively. Thus, taking an offline auto-tuning approach for full applications may not be enough. Finally, in an era when Grid/Cloud computing is getting increasingly popular, runtime adaptation of applications in heterogeneous computing environments is more important than ever.

Development of an online auto-tuner, however, presents its own set of challenges. Managing the cost of the search process and the cost of generating and compiling code-variants on-the-fly are two daunting challenges that must be addressed. Furthermore, the costs of using an online tuning system must be minimal. Otherwise, such costs can overshadow any benefit realized in application performance. If the performance of harmonized code is better (or at least not worse) than that of untuned version of the code, the minimal overhead objective is achieved. Addressing these challenges and making online tuning practical is the topic of this chapter. Our goal is to enable application developers to write applications once and have the auto-tuner adjust the application execution automatically when run on new systems. *Thus, we reach the culmination of the ideas presented so far in this dissertation — we combine many of the ideas presented in the previous chapters to develop a single system that can provide runtime tuning for full programs.* In the next section, we describe our system design.

8.1 System Design

In this section, we describe our online auto-tuning approach for parameters that require new code and present our system design. We divide the discussion into two parts. First, we describe our implementation for code-servers. Second, we present the overall workflow of the runtime auto-tuner.

8.1.1 Code-servers

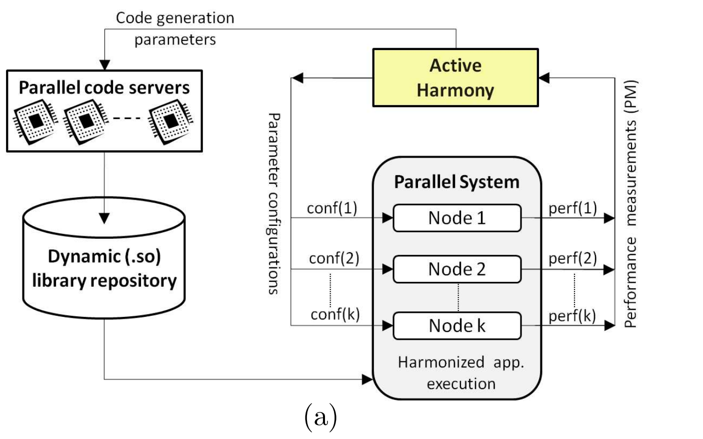
To make runtime auto-tuning practical, the key issue that needs to be addressed is the efficient *runtime* management of the process of generating, compiling, and maintaining a set of alternative implementations and searching among them. A given loop-nest generally requires more than one flavor of transformation strategy. As the number of transformations increases, the number of alternative code-variants grows exponentially. A brute-force approach of generating all possible combinations is, thus, not practical. Instead, our approach generates code variants on-demand by utilizing third-party loop-transformation frameworks.

Active Harmony relies on standalone code-generation utility (or code-servers) for on-demand code generation. Here we describe the two most important features of this utility. First, the design of code-servers allows the users to easily select and switch between available code transformation tools. We separate the search-based navigation of the code-transformation parameter space and the code-generation process, which allows us to easily switch between different underlying code-generation tools (e.g. if we are tuning CUDA code, we can switch to a code-transformation framework that supports GPUs via CUDA or OpenCL)[1]. Second, our code-generation utility can take advantage of idle (possibly remote) machines for distributed code-generation and compilation. Users provide a set of available machines at the start of the tuning session. These machines do the actual code-generation work. Once all code-variants are generated, the compiled code-variants are transported to the scratch filesystem of the parallel machine, where the application being tuned is executing. After the code-generation is complete, our code-generation utility notifies the Active Harmony server about the status.

8.1.2 Overall Workflow

Figure 8.1-(a) shows a schematic diagram of the workflow within our online tuning system. Figure 8.1-(b) shows the application-level view of the tuning process. At each search step, the Active Harmony server issues a request to the code-servers to generate code variants with a given set of parameters for loop transformations. The code-variants that are generated are compiled into a shared library (denoted as `v_N.so` in the figure 8.1-(b)) and placed in a repository (typically on the scratch file systems). Once the code-generation is complete, the application receives a code-ready message from the Active Harmony server. The nodes allocated to the parallel application then load the new code using the `dlopen-dlsym` mechanism. The new code is executed and the measured performance values (denoted as `PM_N` in the figure 8.1-(b)) are consumed by the Active Harmony server to make simplex transformation

[1]For all the experimental results presented in this chapter, we use CHiLL [14], a polyhedra-based compiler framework, to generate code-variants. We discussed the advantages of using CHiLL in section 6.3.1.



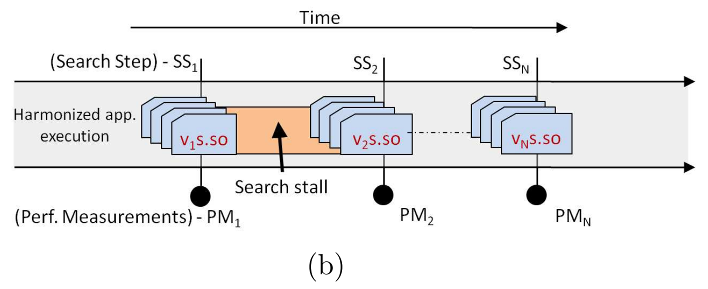


Figure 8.1: Fig.8.1-(a) shows the overall online tuning workflow Fig.8.1-(b) shows application level view of the auto-tuning workflow

decisions. The timing of actual loading of new code is determined by hooks (inserted using the Active Harmony API) in the application. For example, in most programs, we load new code only on timestep boundaries.

Preparing an application for auto-tuning starts with outlining the compute-intensive code-sections to separate functions. We then insert appropriate calls to the outlined functions using function pointers. These function pointers are updated when new codes become available. Currently, the code-sections are outlined manually. In the future, we intend to automate this process using the ROSE compiler framework [54]. Each node running the application keeps track of the best code-variant it has seen thus far in the tuning process. If the code-server fails to deliver new versions on time, the nodes continue their execution with the best version that they have discovered up to that point in the tuning process. The period where no new code is available is referred as `search_stall` (see figure 8.1-(b)). The non-blocking relationship between application execution and dynamic code-generation is important in minimizing the online tuning overhead. The application does not have to wait until the new code becomes available. Furthermore, this asynchronous relationship enables our auto-tuner to exercise control over what code-generation utility to use, how many parallel code-servers to run and how many code-variants to generate in any given search iteration. The policy decisions about what code-variants to generate and evaluate at each iteration is made completely by the centralized tuning server.

8.2 Empirical Results

In this section, we present an empirical evaluation of our framework. First, we conduct a study using as a test application, a Poisson's equation solver program, to determine the least number of parallel code-servers needed to ensure that the `search_stall` phase does not dominate the tuning workflow. Second, we use two parallel applications to demonstrate the effectiveness of our system on three different computing platforms. We compare the performance of harmonized applications with that of original applications compiled with the vendor-suggested highest level of optimization flags turned on. Once the harmonized application is done with its execution, we take the best code-variants returned by the Active Harmony server and run the application using those code-variants. We call these runs `post-harmony` runs.

Active Harmony utilizes the first two search iterations to generate uniformly distributed random configurations from the search space. These configurations are evaluated in parallel. We call these iterations exploratory iterations. The best among these configurations then serves as the starting point for the initial simplex construction. For all our experiments, unroll factors and tile sizes are constrained by the storage capacity of their associated memory hierarchy levels; the machine parameter limits are derived from the CSL description.

To control for performance variability, we use the multiple sampling method. Each configuration is evaluated twice (i.e. the performance of two consecutive timesteps is recorded) and the minimum of the two samples is sent to the Active Harmony server. As discussed in chapter 3 and more fully in our previous work [83], even in the presence of 5% variability due to background noise, taking the minimum of two samples is enough to ensure the convergence of the search algorithm.

8.2.1 Platforms

The experiments were performed on three platforms. The first platform is the umd-cluster (see section 5.2). The second platform (at NERSC [63]) is named Carver, which is an IBM iDataPlex system with 400 compute nodes. Each node contains two quad-core Intel Nehalem 2.67 GHz cores (3,200 cores total in the machine). Nodes are connected via a 4X QDR Infiniband interconnect. These architectures are different from each other not only in terms of the core architectures — the Carver's cores are several generations newer — but also in terms of the interconnect used to connect the nodes. Finally, the third platform, which is named Hopper[2], is a Cray XT5 machine at NERSC. Each node consists of two 2.4 GHz AMD Opteron Shanghai quad-core cores (5,312 cores total in the machine). Nodes are connected via a Seastar2 interconnect.

For the experiments on the umd-cluster, the code-generation and compilation is delegated to idle local machines. For the experiments on Carver, the code-generation is out-sourced to a 64-bit x86 machine at UMD (i.e. code is generated just in time and shipped across the continental United States). This was done because Carver scheduler does not permit synchronized (co-scheduled) jobs yet, which meant that we could not launch a code-generation job simultaneously with the application job. For the experiments on Hopper, the code-generation and compilation takes place on the login nodes.

8.2.2 Calculating the "Net" Speedup

Our runtime tuning strategy uses extra cores to generate and compile new code. Ideally, a fair comparison would be between the execution time of the harmonized application to that of the original application run on $N_h + C$ cores, where N_h is the number of cores the harmonized application is run on and C is the number of cores used for code-generation. However, this is not always possible due to application's data distribution semantics (for example, the application may require the number of cores to be a power of 2). Instead, to account for these extra cores,

[2]Hopper has since been upgraded to a Cray XE6 with 153,408 cores.

we calculate a new metric — *net speedup.* We define charge factor (equation 8.1) as the ratio of the number of cores used to run the application and the total number of cores used for both code-generation and harmonized application execution.

$$c.f. = \frac{N_h}{N_h + C} \tag{8.1}$$

We then multiply the speedup of harmonized applications over the original application by this charge factor to derive the net speedup.

8.2.3 Code-server Sensitivity

With the experimental results presented in this section, we attempt to answer the following question — how many parallel code-servers are needed to ensure that the auto-tuner does not have to wait for too long before the new code is ready? A related question is — How often is the system in the `search_stall` phase? These questions are important because if the `search_stall` phase is long, the application can possibly continue with mediocre parameter configurations for extended periods of time.

The experiments were conducted on the umd-cluster using the PES application (described in section 8.2.4). We controlled the input problem size (1024^3) and the number of cores running the application (128). All 128 cores participate in the tuning process, which means at each search step, code-servers have to generate and compile up to 128 code-variants. This is a typical number of code-variants required per search iteration in all the experiments reported in this chapter. We vary the

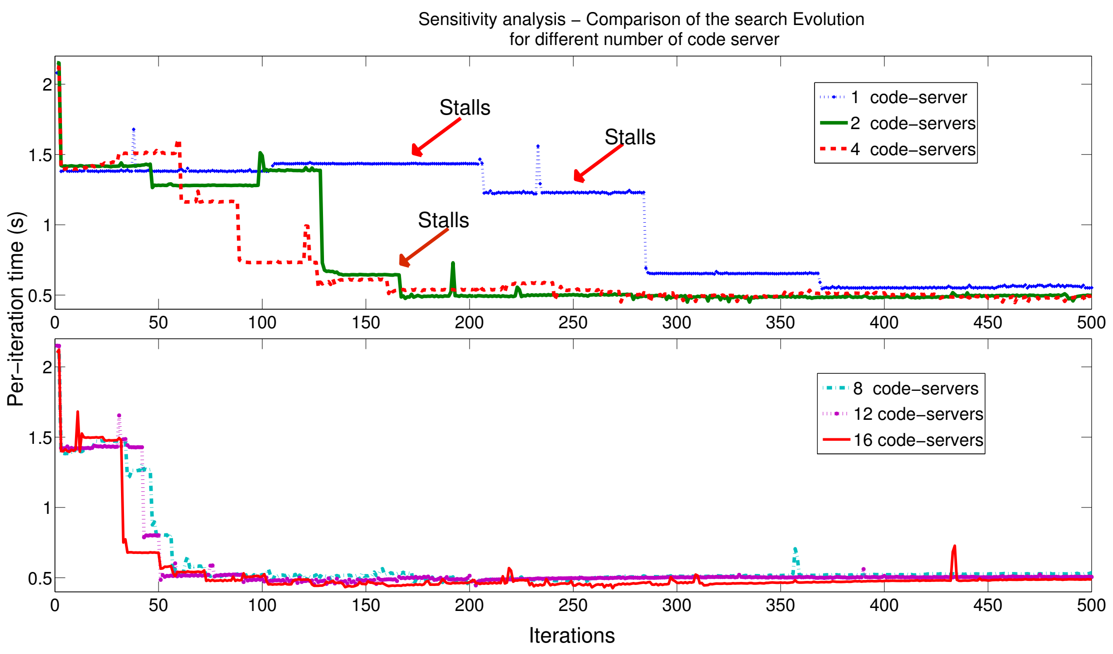


Figure 8.2: Sensitivity results demonstrating how the change in the number of code-servers affects the search evolution

number of code-servers running in parallel and record the average number of time-steps that the application had to continue with the old code. We call this metric "stalled" iterations.

Figure 8.2 shows how per-iteration performance measurements change over time for auto-tuning conducted with alternate number of code-servers. Long stretches of consecutive application timesteps with no performance improvement (marked by arrows) in experiments conducted with 1, 2 and 4 code-servers indicate that the application continued its execution without having new code ready for several timesteps. The same is not true for tuning conducted with 8, 12 and 16 code-servers.

Table 8.1 summarizes the results for this experiment. Consider columns 3 and 5. As the number of code-servers is increased from 1 to 4, the average number of stalled iterations goes down significantly. This is to be expected. What is surprising

Table 8.1: Sensitivity experiment results

# of code servers	Avg. # of search iters	Avg. # of stalled iters	Avg. # of code evaluated	Avg. speedup
1	6*	46	502	0.75
2	17*	13	710	0.97
4	27	7.18	928	1.04
8	23	4.48	818	1.23
12	22	4.06	833	1.21
16	26	3.59	931	1.24

* - search algorithm did not converge

is that the addition of extra code-servers from 8 to 16 does not significantly change the application speedup or the number of stalled iterations. The reason for this is that as the search algorithm evolves and starts converging to a point in the search space, the load on code-servers goes down (i.e., more points in the simplex become identical). The data in column 4 of table 8.1 shows the number of unique code-variants evaluated in different experiments. We can see that as the average number of search iterations goes from 6 for runs using 1 code-server to 17 for runs using 2 code-servers, the average number of unique code-variants goes up by only 208.

Our end goal with this experiment was to set a minimum number of code-servers required to ensure a short `search_stall` phase for the rest of the experiments. Having said that, we acknowledge that there are other factors that can play important roles in setting this minimum. In one of our preliminary experiments, we discovered that the size of the search space can also dictate this minimum. When we ran the experiments with small tuning space, the exploratory iterations (initial random search of points) were able to find good parameter configurations. In this case, the number of stalled iterations did not matter because applications were already executing with good configurations. Moreover, the number of minimum code-servers can (and most probably will) change if we switch between different code-generation tools. Currently, we are looking into more robust ways to account for these issues and to derive the value for the minimum required parallel code-servers. For the rest of the auto-tuning experiments we describe in this chapter, we used 8 code-servers.

8.2.4 Subject Application: Poisson's Equation Solver (PES)

Poisson's equation is a partial differential equation that is used to characterize many processes in electrostatics, engineering, fluid dynamics, and theoretical physics. To solve for Poisson's equation on a three-dimensional grid, we use a modified version of the parallel implementation provided in the KeLP-1.4 distribution [6]. The application is written in C++ and Fortran. The implementation uses the redblack successive over relaxation method to solve the equation. The core of the computational time is spent on the relaxation function, which uses a 7-point stencil operation, and the error calculation function, which calculates the sum of squares of the residual over the 3D grid. These two code-sections are tuned simultaneously using the Active Harmony framework. The code-sections are outlined in table 8.2.

The original implementation of the relaxation operation uses the first half of one iteration to update "red" array points and the second half of the iteration to update the "black" points. In the adapted version of the application, we use the fused version of the relaxation operation. The fused version orders the loop iteration so that black points in each column are updated immediately after the red points

Table 8.2: PES tuning: top table shows the kernels. Bottom table provides the transformation recipes and constraints

Kernel	*Original Code*

Relaxation	`do kk=wl2-1,wh2`
	` do k=kk+1,kk,-1`
	`   if ((k.le.wh2).and.`
	`       (k.ge.wl2)) then`
	`     do j=wl1,wh1`
	`       do i=wl0+mod(kk+j+1,2),wh0,2`
	`         u(i,j,k) = c * (u(i-1,j,k)`
	`          + u(i+1,j,k) + u(i,j-1,k)`
	`          + u(i,j+1,k) + u(i,j,k-1)`
	`          + u(i,j,k+1)- c2*b(i,j,k))`

L2 Norm (Error)	`do k = 1, N`
	`  do j = 1, N`
	`    do i = 1, N`
	`      du = c*(u(i-1,j,k) + u(i+1,j,k)`
	`            + u(i,j-1,k) + u(i,j+1,k)`
	`            + u(i,j,k-1) + u(i,j,k+1)`
	`            - c2*u(i,j,k))`
	`      r = b(i,j,k) - du`
	`      err = err + r*r`

Kernel	*Transformation Recipe*	*Constraints*

Relaxation	`Manual tiling:`
	` i and j loops`
	` (TI1, TJ1)`
		$TI1 \times TJ1 \leq \frac{1}{2}\left(\frac{cache\_size}{2}\right)$
		$TI1 \geq TJ1$
		$TI1 \in [0, 4, \ldots, prob\_size]$
		$TJ1 \in [0, 4, \ldots, prob\_size]$

L2 Norm (Error)	`original()`
	`tile(0, 3, TI2)`
	`tile(0, 3, TJ2)`
	`tile(0, 3, TK2)`
	`unroll(0, 6, UI2)`
		$TI2 \in [0, 4, \ldots, prob\_size]$
		$TJ2 \in [0, 4, \ldots, prob\_size]$
		$TK2 \in [0, 4, \ldots, prob\_size]$
		$TI2 \geq TJ2$
		$TJ2 \geq TK2$
		$UI2 \in [1, 2, \ldots, register\_size]$
		
		Search space dimension : 6 *Parameters* :
		$[TI1, TJ1, TI2, TJ2, TK2, UI2]$
		
		Sample search space size: 3.11×10^7
		possible configurations for
		$64 - core\ run\ with\ 512^3 domain - size$

in the next column and vice versa [76]. This fused version (see table 8.2) serves as the baseline for comparing the net speedup of harmonized PES.

Our tuning strategy for this application combines symbolic parameter tuning[3]

[3]Symbolic tuning refers to tuning for parameters that are symbolic, i.e. no new code is necessary to move between parameter values.

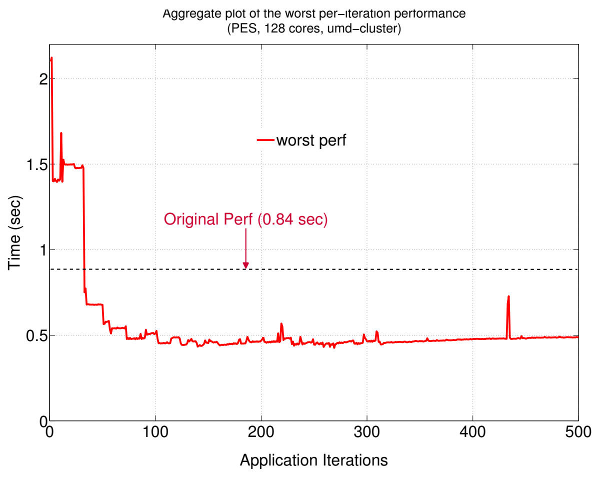


Figure 8.3: A plot for aggregate worst timing at each iteration

and tuning with dynamic code-generation. For the relaxation function, we use symbolic tuning. We tile the two outermost loops and use Active Harmony to determine the dimension of the tiles. The error function is optimized using the dynamic code-generation method. For this function, we tile all three loops and the innermost loop is unrolled. The search space is, thus, six-dimensional (two tunable parameters for the relaxation function and four for the error function). All cores allocated to the application participate in the tuning process. Thus, a 128-core run of this application evaluates up to 128 tiling configurations simultaneously for the relaxation function and up to 128 loop-variants simultaneously for the error function in a single search step. The optimization strategy (expressed in terms of the CHiLL recipe) along with the constraints for unbound tunable parameters is provided in table 8.2. For a 512^3 problem size run on 64-cores, the search space has approximately 3.11×10^7 possible configurations.

We performed two sets of auto-tuning experiments — one using 64 cores and one using 128 cores. Both sets of experiments were done on the umd-cluster. For

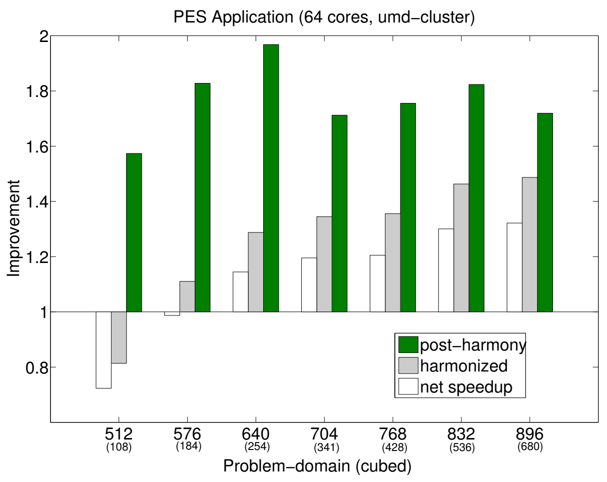


Figure 8.4: Performance improvement of harmonized PES, net speedup, and post-harmony run of the solver (64 core run on umd-cluster)

each core count, we select multiple input domain sizes. Figure 8.3 shows how Active Harmony steers per-iteration performance of the harmonized PES. This experiment uses a grid size of 1024^3. The figure plots the timing of the worst performing configuration for each application iteration. The running time of an SPMD-based application is bounded, at each timestep, by the slowest configuration. Per-iteration time for the original application is indicated by the horizontal line in the figure. The figure shows that Active Harmony suggested configurations outperform the original application's per-iteration timing within the first few tens of iterations.

Figures 8.4 and 8.5 plot the net and harmonized speedups achieved within one full execution of the harmonized PES. The original application execution times (in seconds) are shown in parentheses below the label for x-axis. The application was run on 64 and 128 cores on the umd-cluster for varying input data sizes. As expected, as the size of the problem domain increases, the performance of the harmonized application increases as well. This is intuitive because with the increase in

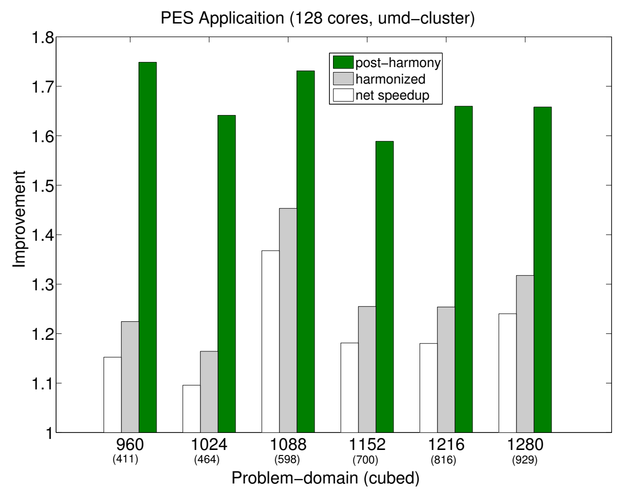


Figure 8.5: Performance improvement of harmonized PES, net speedup, and post-harmony run of the solver (128 core run on umd-cluster)

the problem size, the Active Harmony server gets more time to explore the search space before the application completes its execution. For the 512^3 problem size (see figure 8.4), the program execution time is too short (108 seconds) and the harmonized application runs 28% slower than the original untuned version. The 28% slowdown does incorporate the charge factor for 8 extra cores used for code-generation. The harmonized application is unable to overcome the penalty of using some poor configurations early in the short run of the program (132 seconds elapsed time).

On average, for both core counts and different problem domains, harmonized PES, in terms of the net speedup, performs 1.16 times faster than the original application. Thus, even after allowing for the code-servers, a single execution with no prior runs is, on average, 16% faster than the original application. The best net speedup for the harmonized application is 1.37. Post-harmony runs, which use Active Harmony suggested parameter configurations and code-variants, on average, perform 1.72 times faster than the original application. This indicates the performance gain if the program was run a second time on the same machine with similar inputs.

8.2.5 Subject Application: Parallel Multiblock Lattice Boltzmann (PMLB)

The Lattice Boltzmann Method (LBM) is a widely used method in solving fluid dynamic systems. In contrast to the conventional methods in fluid dynamics, which are based on the discretization of macroscopic differential equations, the LBM has the ability to deal efficiently with complex geometrics and topologies [92]. For our experiments, we use the parallel multiblock implementation (extended to 3D problems) of the LBM developed by Yu et al [96]. The test case lattice model for our experiments is `D3Q19` (19 velocities in 3D) with the collision and streaming operations. The application is written in C.

The PMLB code is divided into six main operations: initialization, collision, communication, streaming, physical and finalization. Collision, communication, streaming and physical operations are executed within a loop. Initialization and finalization operations are performed once. We focus our attention on the streaming operation, which accounts for more than 75% of the execution time. The streaming operation moves particles in motion to new locations along with their respective 19 velocities. This operation requires a significant number of memory copy operations.

The streaming operation consists of five separate triply-nested kernels, which

Table 8.3: PMLB tuning: top table shows the kernel. Bottom table provides the transformation recipe and constraints

Kernel	*Original Code*

streaming 1	`for (i=1; i<=imax;i++)`
	`  for(j=1; j<=jmax; j++)`
	`   for(k=1; k<=kmax; k++)`
	`    {`
	`      c1 = i*(ny_local);`
	`      c2 = c1+(ny_local);`
	`      c3 = (c1+j)*(nz_local);`
	`      c4 = c3+(nz_local);`
	`      c5 = (c2+j)*(nz_local);`
	`      c6 = c5+(nz_local);`
	`      c7 = (c3+k)*en;`
	
	`      fi[ 6+c7]=fi[6+(k+1+c3)*en];`
	`      fi[ 4+c7]=fi[4+(k+c4)*en];`
	`      fi[18+c5]=fi[18+(k+1+c4)*en];`
	`      fi[ 2+c7]=fi[ 2+(k+c5)*en];`
	`      fi[14+c7]=fi[14+(k+1+ c5)*en];`
	`      fi[10+c7]=fi[10+(k+c6)*en];`
	`    }`

Kernel	*Transformation Recipe*	*Constraints*

*streaming*1	`original()`
	`tile(0, 1, TI)`
	`tile(0, 3, TJ)`
	`unroll(0, 5, UK)`
	`known(imax>1)`
	`known(jmax>1)`
	`known(kmax>1)`
		$TI \in [0, 4, \ldots, prob\_size]$
		$TJ \in [0, 4, \ldots, prob\_size]$
		$UK \in [1, 2, 3, 4]$
		$TI \geq TJ$
		
		Search space dimension : 6
		2 *sets of* $[TI, TJ, UK]$:
		one for fused kernels and one for non−fused kernels
		
		Sample search space size: : 8.92×10^6 *possible configurations for* $128 - core$, 512^3 *problem size*

are tuned simultaneously. Our optimization strategy utilizes loop-fusion, loop-tiling and loop-unrolling. The tuning is done in two phases. The first few iterations of the LBM method are used to identify the best fusion configuration for the five triply nested loops within the streaming operation. For this stage, we use the exhaustive search. Once we identify the best performing fusion configuration, the tuning moves to the second stage, which involves tiling the outermost two loops

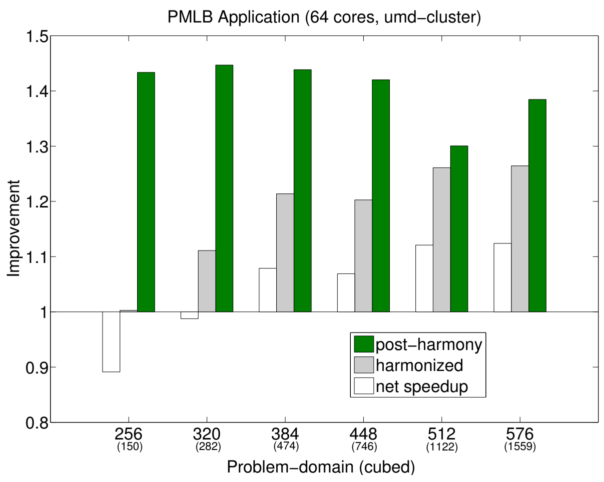


Figure 8.6: Performance improvement of harmonized PMLB, net speedup, and post-harmony run of PMLB (64 core run on umd-cluster)

and unrolling the innermost loop[4]. The second stage uses the parallel rank ordering algorithm to determine two sets of tiling and unrolling factors — one for the fused loop-nests and another for the remaining loop-nests. The search parameter space for all PMLB experiments is, thus, six-dimensional. The optimization strategy (expressed in terms of the CHiLL recipe) along with the constraints for unbound tunable parameters is provided in table 8.3. Out of the five deeply nested kernels in the streaming operation, we show only one kernel in the table. Other kernels are similar in structure. For a 512^3 problem size run on 128-cores, the search space has approximately 8.92×10^6 possible configurations.

PMLB tuning experiments were done on the umd-cluster, Carver and Hopper. Figures 8.6 and 8.7 plot speedup results for harmonized and post-harmony PMLB runs using 64 and 128 cores on umd-cluster. We use multiple input datasets for different core counts. Again, we see that the increase in the size of the problem domain

[4]Simple code modifications were required to remove scalar dependencies between different levels of loop-nests to ensure legality of code transformations.

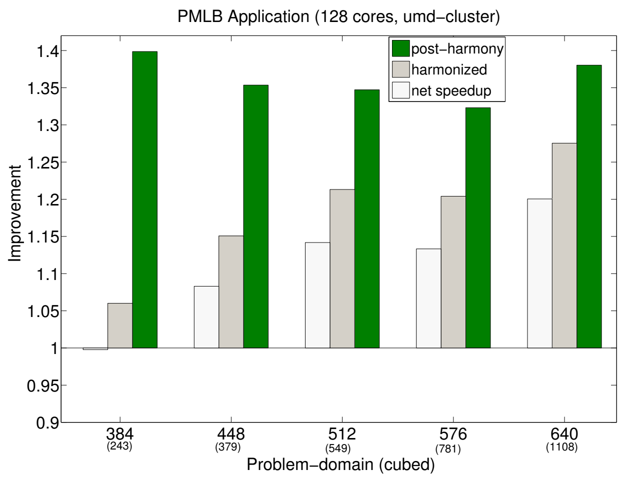


Figure 8.7: Performance improvement of harmonized PMLB, net speedup, and post-harmony run of PMLB (128 core run on umd-cluster)

leads to a better performance for the harmonized PMLB. On average, harmonized PMLB performs 1.14 times faster than the original application, while post-harmony runs perform 1.38 times faster than the original application.

For the experiments on Carver, we use two different core counts — 256 and 512. We were limited in terms of the number of core counts because 512 is the maximum core count a user can reserve on Carver. Figures 8.8 and 8.9 plot speedup results for harmonized and post-harmony runs using 256 and 512 cores of Carver. In terms of the net speedup, on average, harmonized PMLB performs 1.11 times faster than the original application. The best net speedup for a harmonized run is 1.46, i.e. even after factoring in the extra cores for code-generation, a single execution of harmonized PMLB is up to 46% faster than the original application. Post-harmony runs perform, on average, 1.37 times faster than the original application.

Experiments on Hopper were done using 512 and 1024 cores. Figures 8.10 and 8.11 plot speedup results for harmonized and post-harmony runs using 512 and

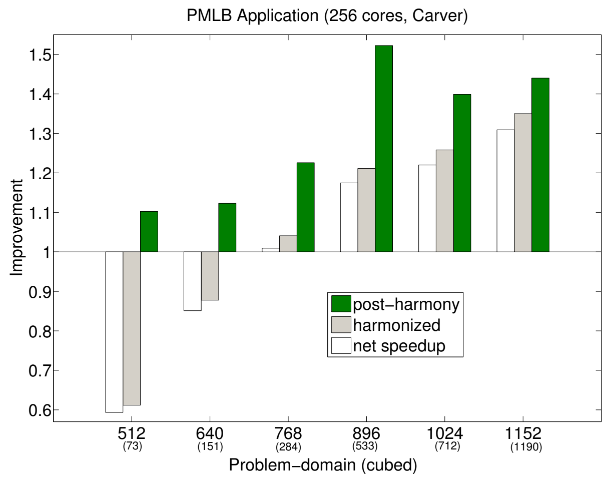


Figure 8.8: Performance improvement of harmonized PMLB, net speedup, and post-harmony run of PMLB (256 core run on Carver)

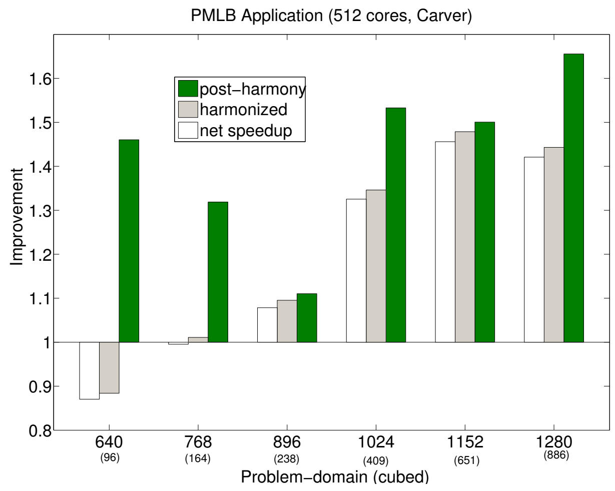


Figure 8.9: Performance improvement of harmonized PMLB, net speedup, and post-harmony run of PMLB (512 core run on Carver)

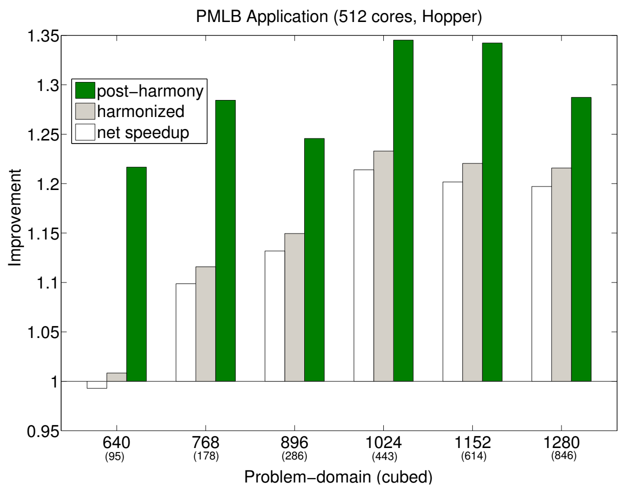


Figure 8.10: Performance improvement of harmonized PMLB, net speedup, and post-harmony run of PMLB (512 core run on Hopper)

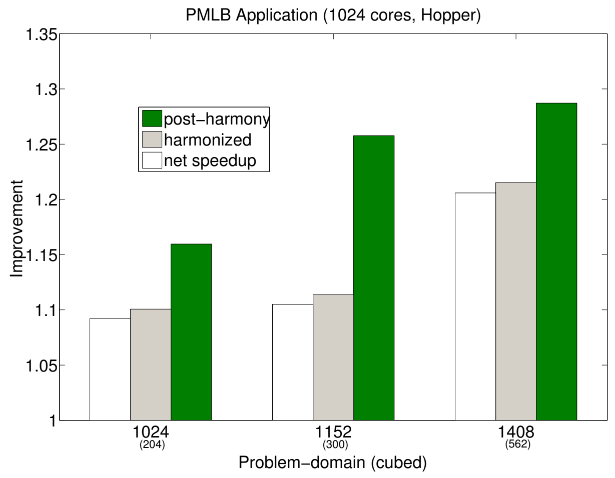


Figure 8.11: Performance improvement of harmonized PMLB, net speedup, and post-harmony run of PMLB (512 core run on Hopper)

Table 8.4: Results for cross-platform experiments

Problem-size	**speedups for post-harmony on umd-cluster**		
	w/ umd confs	w/ Carver confs	w/ Hopper confs
448^3	1.42	1.13	1.00
512^3	1.30	1.26	0.95
576^3	1.38	1.16	1.02

Problem-size	**speedups for post-harmony on Carver**		
	w/ Carver confs	w/ umd confs	w/ Hopper confs
448^3	1.51	1.38	1.34
512^3	1.34	1.31	1.33
576^3	1.42	1.39	1.27

Problem-size	**speedups for post-harmony on Hopper**		
	w/ Hopper confs	w/ Carver confs	w/ umd confs
448^3	1.28	1.30	1.27
512^3	1.34	1.31	1.28
576^3	1.31	1.35	1.30

1024 cores of Hopper. In terms of the net speedup, on average, harmonized PMLB performs 1.14 times faster than the original application. The best net speedup for a harmonized run is 1.21. Post-harmony runs perform, on average, 1.28 times faster than the original application.

8.2.6 Cross-platform Comparison

In the preceding section, we showed how we applied Active Harmony to auto-tune the execution of two parallel applications on three different platforms. A logical question is how do the parameters that Active Harmony selected for different platforms relate to each other. To answer this question, we conducted a controlled study using 64-cores on all three systems. We selected three problem sizes for the PMLB application. The selection of the problem sizes was based on whether Active Harmony's search converges to a solution or not within a single execution of the harmonized PMLB. This was done to ensure that Active Harmony gets a fair chance to select good configurations on all the systems. We then use Active Harmony suggested parameter configurations for a given problem size on one system to conduct post-harmony runs for the same problem size on other systems.

The results are summarized in table 8.4. Post-harmony runs conducted on the umd-cluster using the configurations suggested for Carver do perform better than the original version of the application. However, the speedup difference between post-harmony runs conducted on the umd-cluster with umd-cluster configurations and Hopper configurations is rather significant. Upon closer look at the parameter values, we observed that for umd-cluster and Carver, Active Harmony only fuses the first and the third kernels in the streaming operation. While on Hopper, the first, the third and the fifth kernels are fused together. Furthermore, on Hopper, the second and the fourth kernels are fused as well. We suspect that the poor performance can be attributed to the properties and size of the instruction cache on umd-cluster. It is also possible that excessive application of loop-fusion causes more register spills on umd-cluster than for the other two platforms, thereby degrading the performance of the PMLB [73]. This result argues tuning not only for specific architecture but also for specific processor implementation. In the future, we plan to look at the question of how the configurations found for different input datasets for the same harmonized application relate to each other.

It is also interesting to see that the post-harmony runs conducted on Carver using Hopper configurations provide similar speedups when compared to the post-harmony runs conducted on Carver using Carver configurations. The same is true for post-harmony runs conducted on Hopper. We attribute this to the processor architecture similarity of the two systems.

In the work presented in this chapter, we focused exclusively on optimizing computation at the core level. In the future, we plan to look into communication auto-tuning in tandem with per-core tuning. We believe that the difference in the interconnect technology between the Carver and Hopper systems will show the benefits of tuning for specific interconnect technology.

8.3 Summary

In this chapter, we presented our runtime compilation and tuning infrastructure designed to improve the performance of parallel applications within a single execution. Since the system does not rely on any specific code-generation system, new code transformations can be easily incorporated within our system. We showed that for two programs, auto-tuning improves performance without training runs.

Even if the intent is to auto-tune an application for a specific machine and leave it fixed, runtime code-generation is useful. By generating and trying multiple configurations in a single run, we greatly reduce the time required to auto-tune a program.

Our system enables application developers to write applications once and have the auto-tuner adjust the application execution dynamically when run on new systems. We demonstrated the value of our system by applying it on real application codes. The performance improvement of up to 46% for a 512-core parallel application execution can be achieved within a single execution of the application.

Chapter 9

Future Work

There are several natural extensions of our auto-tuning framework. We divide the discussion into two parts. The first part describes the short-term vision for Active Harmony and the second part describes the long-term vision.

9.1 Short-term Vision

In this section, we present a couple of ideas that can have near-term impact in improving the usability and performance of Active Harmony. These ideas are the result of experience gained in the implementation of Active Harmony and our recent discussions with Active Harmony users.

9.1.1 Code-tuning API

The current implementation of Active Harmony provides multiple examples that demonstrate how the end-users can utilize our online adaptive code-generation and tuning feature. As the system matured, we realized that these examples show remarkable commonalities in terms of the overall tuning workflow. These commonalities can be extracted into a code-tuning API to reduce the amount of code modifications developers have to make to harmonize their programs.

9.1.2 Online Tuning for AMR Codes

Active Harmony's online tuning capability can help Adaptive Mesh Refinement (AMR) codes. Rather than relying on one global grid resolution, AMR codes have the ability to change the underlying granularity of the mesh or grid locally during a single production run of the application [55]. Areas in the domain that need finer grid resolution (e.g. area near the heat source in heat diffusion problem) can benefit from AMR technique because this allows shifting of the computational resources to the parts of the domain that need these resources the most. This dynamic change in the mesh structure also changes the execution characteristics of the application. Thus, an offline auto-tuner cannot adequately address the auto-tuning needs for AMR codes. Instead, Active Harmony's online adaptive code-generation and tuning is better suited to tune AMR codes. Our runtime auto-tuner can help AMR codes react to the changes in workloads and suggest different code-variants based on the grid resolution.

9.1.3 Code-server Sensitivity

In chapter 8, we described our methodology to find a minimum number of code-servers required to ensure a short `search_stall` (see section 8.1) phase for the runtime auto-tuning experiments presented in that chapter. We also acknowledged that there are other factors that can play important roles in setting this minimum value. Some of these factors include the size of the search space, the underlying code-generation utility and the compiler, the number of cores the harmonized application is running on, etc. A model-based approach that takes into consideration the factors that we just mentioned to find the minimum required number of code-servers will help in further shortening the `search_stall` phase.

Furthermore, we would like to extend the distributed code-generation utility so that we can add additional resources when the code-generation load is heavy and remove the resources when the load is light.

9.2 Long-term Vision

In this section, we present some long-term ideas for Active Harmony. We discuss the likely future trends in the auto-tuning research. We also describe how Active Harmony can play a significant role in addressing the auto-tuning challenges posed by the next generation exascale systems.

9.2.1 Power Auto-tuning

As we are entering the era of exascale systems, the key problem that the HPC community is trying to address is the "power wall" problem. The problem arises from the fact that as compute nodes (consisting of multi/many-cores) become increasingly powerful, they also become increasingly power-hungry. The problem is further exacerbated by the fact that these cores do need to be cooled down as well. Going forward, we see the power-aware computing research in the HPC community focus in two main areas. The first area of research will consist of projects that are involved in developing simple and low-cost hardware and software solutions to construct the power consumption profile of scientific applications. Devices such as PowerMon2 [7] and software frameworks such as PowerPack [29] can help measure an application's impact on CPU, memory, hard-disk and peripheral bus power consumption. The second area of research will be led by auto-tuners that utilize the information provided by the first to automatically generate code-variants that reduce the power foot-print of different code-sections.

Active Harmony is well positioned to make contributions to the second area of research. An obvious starting point is to redefine the objective function from application-level performance (e.g. execution time, cache hits) to a system-level metric that captures power consumption (e.g. FLOPS/watt). We can then use the compiler-based auto-tuning design presented in this thesis to find code-variants that reduce power consumption.

We expect application developers and library writers to implement and evaluate alternative implementations of key algorithms (e.g. data distribution, collective communication) to make their code ready for exascale systems. Each of these alternatives can have drastically different power consumption and performance profiles. For example, an implementation which aims at reducing the power consumption by limiting inter-node communication can increase computational load at core-level. Active Harmony can help developers quantify this tradeoff and make choices that balance application performance and power consumption. Active Harmony can also help limit the load on some overused resource (e.g. interconnect) by switching to algorithms that reduce the load on the resource.

Finally, the auto-tuner can also help reduce power consumption by suggesting selective reduction in operating voltage of unused electronic components of a system. It is reasonable to have different instances of Active Harmony monitoring an application's use of different key electronic components. If the component is underutilized for extended periods of time, the component can be powered down. This strategy can, however, have undesirable consequences (e.g. application can fail when it needs a component that is turned off or powered down). A careful combination of information from historical power profiles (which can alert the system to power up the components before the application needs it), and robust checkpointing/restart mechanism can help avoid and mitigate those undesirable consequences.

9.2.2 Communication Tuning for Exascale Systems

In this thesis, we have focused mostly on improving node-level performance. Inter-node communication is another critical factor that needs our attention. For applications running on large number of nodes, inter-node communication can quickly become the most dominant factor in the overall application execution time. Exascale systems are projected to have thousands of nodes (with thousands of cores on each node). Communication tuning will therefore be crucial in expanding the scope and usability of Active Harmony.

Towards that end, one interesting research area for Active Harmony is to explore communication level parameters such as communication and computation overlapping strategies and interconnect-specific parameters. Load balancing strategies also play a vital role in reducing communication costs. Over the course of our research, we have found frequent examples where even a slight load imbalance can increase the communication costs by significant margins. Active Harmony can help developers choose between different data-distribution algorithms during training runs. If dynamic data redistribution methodologies are made available by the application (e.g. adaptive mesh refinement applications), Active Harmony can trigger redistribution when the communication costs exceed a certain threshold.

We believe that hybrid MPI-OpenMP programming model will be extensively used by programmers to port current applications to the next generation exascale systems. We attribute this to the ever increasing number of cores in compute-nodes and to the fact that intra-node communication is much cheaper than the inter-node communication. Therefore the need for an auto-tuner that can explore the combined search space of MPI and OpenMP parameters cannot be overstated. Active Harmony can help find a performance enhancing balance between MPI and OpenMP by selecting appropriate parameters (e.g. chunk-size, number of OpenMP threads, MPI buffer size) for hybrid programs. These parameters can also be selected dynamically to react to the changing behavior of the application and the computing platform.

9.2.3 CHiLL Recipe Library

In this dissertation, we selected a single parametrized loop transformation recipe for each loop-nest that we tuned. In the future, we would like to explore the idea of the auto-tuner automatically selecting a recipe from a library of recipes —

a library that consists of platform-specific recipes for the most commonly seen loop structures (e.g. stencil operations, dense and sparse linear algebra kernels, streaming computations) in scientific applications. This library can be created and evolved by compiler experts, application developers and auto-tuners based on their experience working with real codes. Finally, we envision an auto-tuner that is capable of generating recipes for arbitrary program components and loop-nests.

9.2.4 CSL Library

The idea of a CSL library is analogous to that of the CHiLL recipe library. Contributions to the library can come from auto-tuners, application developers, compiler experts, performance models and also from the platform vendors. The library will consists of two parts. The first part will consist of platform-specific CSL descriptions for the most commonly seen loop structures in scientific applications. This description will describe the parameter space, tuning strategies and constraints and relationships between the tunable parameters. The second part will provide machine models. The models will incorporate architecture-specific details such as memory bandwidth and latency, memory hierarchy information etc.

Chapter 10

Conclusion

In this dissertation, we described a unified end-to-end solution to auto-tuning parallel applications. Our system is scalable, general-purpose and provides tuning mechanisms for all stages of application development and deployment — compile time, application launch time and runtime. The search-based tuning system is empowered with a parallel parameter tuning algorithm, which can take advantage of the available parallelism inherent in today's High Performance Computing systems. The empirical results presented in this document showed that our tuning algorithm can effectively deal with high-dimensional search spaces. The fact that the search algorithm converges to solutions in only a few tens of search-steps while simultaneously tuning multiple parameters demonstrates its capability of taking into account the latent interactions between tunable parameters.

We studied the nature of performance variability in real systems. One of the most important observations we made was that the performance variability is heavy-tailed. Heavy-tailed performance (execution time) distributions consist of many small spikes with random occurrences of few large spikes, most likely because of external factors. What this means from the perspective of auto-tuning is that there is a non-negligible probability of observing large variations in the measured/estimated performance. The sampled performance measurements could, therefore, have infinite variance. This observation essentially renders the average operator, which is the most widely used operator to get an estimate for “real” performance from multiple samples, ineffective. As an alternative, we suggest taking a minimum of multiple performance measurements. The minimum has finite mean and variance and is not heavy-tailed.

We evaluated our auto-tuner using real applications and benchmark kernels. Our system leverages available compiler technology to generate code on-the-fly for tunable parameters that require new code. We showed how our runtime compilation and tuning methodology improves the performance of parallel applications within a single execution. Since the system does not rely on any specific code-generation system, new code transformations can be easily incorporated within our system. We showed that for multiple large-scale parallel programs, auto-tuning improves performance without training runs.

Furthermore, our system enables programmers to write applications once and have the auto-tuner adjust the application behavior automatically when run on new systems or on the same system with a new workload. The performance improvement of up to 46% for a 512-core parallel application execution can be achieved within a single execution of the application. For a 1024-core execution of the same application, our system improved the execution time by 21%. Even if the intent is to auto-tune an application for a specific machine and leave it fixed, our system is useful. By generating and trying multiple configurations in a single run, we greatly reduce the time required to auto-tune a program.

The success of any auto-tuning research is largely determined by whether it can successfully steer application's performance. The overhead of using an auto-tuner should be minimal and the auto-tuning infrastructure should be scalable. We showed that our system shows remarkable promise in all these fronts. Active Harmony brings together all tunable targets in a given application within a single unified auto-tuning framework. Entities within an application (e.g. library parameters, computationally intensive loop nests) that can be changed or transformed (for better performance) without affecting the application result are appropriate for exploration by Active Harmony. Therefore our auto-tuner is general-purpose and the system can be used to tune compiler-level parameters, application-level input parameters and runtime parameters.

Appendix A

Constraint Specification Language

Tables A.1 and A.2 provide the Extended Backus Naur Form (EBNF) grammar for the Constraint Specification Language. The syntax for expressions is adapted from an expression evaluator example written in ANTLR [77]. This expression syntax closely resembles the C (and other high-level language) EBNF syntax. Table A.3 provides an example specification for Matrix Multiplication tuning using the CSL. Table A.4 is the output of the standalone tool that parses the CSL specification and outputs a python script. The python script uses the python-constraint solver module to enumerate the legal points in the search space.

We use the following convention to present the CSL grammar in tables A.1 and A.2. Symbols and strings that appear as a part of the parameter specification are "double-quoted" (e.g. "search", "+"). Unquoted curly brackets (i.e. `{` and `}`) are used for better readability of the grammar. Symbols `?`, `*` and `+` have the following meaning:

`?`: Symbol (or a group of symbols in curly brackets) can appear zero or one time.

`*`: Symbol (or a group of symbols in curly brackets) can appear zero or multiple times.

`+`: Symbol (or a group of symbols in curly brackets) has to appear at least once and can appear multiple times.

Table A.1: Constraint Specification Language grammar - Part I

```
<parameter space> ::= "search" "space" <parameter space name>
                      "{"
                          <space body>
                      "}"

<space body> ::=       {<constant declaration>}*
                       {<code region declaration>}*
                       {<region set declaration>}*
                       {<parameter declaration>}+
                       {<constraint declaration>}*
                       {<constraint specification>}*
                       {<grouping info>}*
                       {<ordering info>}*

<parameter space name> ::= <identifier>
<identifier> ::= ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*

<csl comment> ::= '#' (~('\n'|'\r'))* ('\n'|'\r'('\n')?)

Constant Declaration Part:
--------------------------------------------
<constant declaration> ::= "constants" "{" {<constants>}+ "}"

<constants> ::= <param type> <constant name> "=" <domain val> ";"

<constant name> ::= <identifier>

Code Region and Region Set Declaration Part:
--------------------------------------------
<code region declaration> ::= "code_region" <region name> ";"

<region name> ::= <identifier>

<region set declaration> ::= "region_set"
                           <region set name> "[" <region name> <region name list> "]"";"

<region set name> ::= <identifier>

<region name list> ::= <region name> { "," <region name> }*

Parameter Declaration Part:
--------------------------------------------
<param declaration> ::= "parameter" <parameter name> <param type>
                        "{"
                            <domain restrictions> {<default spec>}? {<region spec>}?
                        "}"

<parameter name> ::= <identifier>

<param type> ::= "int" |  "float" |  "string" |  "bool"  |  "mixed"

<domain restrictions> ::= "range"  "["<domail val> ":" <domain val> (":" <domain val>)? "]" ";"
                       |  "prange" "["<domail val> ":" <domain val> ":" <domain val> "]" ";"
                       |  "array"  "[" array ("," "[" array "]")* "]" ";"

<array> ::= <domain val> ("," <domain val>)*

<domain val> ::= <integer> |  <float> |  <string>  |  <boolean>

<default spec> ::= "default" <integer> ";" | "default" <float> ";"  | "default" <string> ";"
                | "default"  <boolean> ";"

<boolean> ::= "T" | "F"
<integer> ::= <digit> | <integer> <digit>
<float> ::= <integer> "." <integer>

Continued in Table A.2.
```

Table A.2: Constraint Specification Language grammar - Part II

```
Continuation from Table A.1.

<digit> ::= [0-9]
<string> ::=  '"' ( 'a'..'z' | 'A'..'Z' | '_' | '0'..'9' | ' ' )* '"'

<region spec> ::= "region" <region set name> ";"

Constraint Declaration Part:
--------------------------------------------
<constraint declaration> ::= "constraint" <constraint name>
                             "{" <expression> ";" "}"

<constraint name> ::= <identifier>

<expression> ::= <logical expression>

<logical expression> ::= <boolean and expression> {"||" <boolean and expression> }*

<boolean and expression> ::= <equality expression> {"&&" <equality expression>}*

<equality expression> ::= <relational expression> {("="|"!=") <relational expression>}*

<relational expression> ::= <additive expression> {("<"|"<="|">"|">=") <additive expression>}*

<additive expression> ::= <multiplicative expression> {("+"|"-") <multiplicative expression>}*

<multiplicative expression> ::= <power expression> {("*"|"/"|"%") <power expression>}*

<power expression> ::= <unary expression> {"^" <unary expression>}*

<unary expression> ::= <primary expression> | "!" <primary expression>
                    |  "-" <primary expression>

<primary expression> ::= "(" <logical expression> ")" |  <domain val>
                      |  <parameter reference>

<parameter reference> ::= <identifier> | <identifier> "." <identifier>
                       |  <identifier> "." "value"

Bind everything together with specification:
--------------------------------------------
<constraint specification> ::= "specification"
                               "{"
                                   <specification expr>";"
                               "}"

<specification expr> ::= <primary spec expr> (("&&"|"||") <primary spec expr>)*

<primary spec expr> ::= "(" <specification expr> ")" | <constraint name>

Parameter Ordering:
--------------------------------------------
<ordering info> ::= "ordering"
                    "{" <parameter list> "}" ";"

<parameter list> ::= <parameter name> | <parameter name> {"," <parameter name>}+

Parameter Grouping:
--------------------------------------------
<grouping info> ::= "groups"
                    "{" {<set declaration>}+ "}"

<set declaration> ::= "set" "[" <parameter reference> {"," <parameter reference>}* "]" ";"
```

Table A.3: Search specification for MM tuning (tiling and unrolling)

```
search space tiling_mm {
# Now defining the search space specifics for tiling and unrolling
# define some constants
constants {
    int l1_cache=128;
    int l2_cache=4096;
    int register_file_size=16;
}
# code region declarations: loopI, loopJ, loopK
code_region loopI;
code_region loopJ;
code_region loopK;
# region set declaration
region_set loop [loopI, loopJ, loopK];

# declare tile_size parameter and associate the parameter to region
# set loop. default value of the parameter is set to 32.
parameter tile int {
    # prange -> power range [min:max:base]
    prange [1:8:2];
    # 2^1, 2^2, ..., 2^8
    default 32;
    region loop;
}
# declare unroll_factor parameter and associate the parameter to
# region set loop. default unroll factor is set to 1.
parameter unroll int {
    range [1:8:2];
    default 1;
    region loop;
}
# L1 cache (for array B)
constraint mm_l1 {
    loopK.tile * loopJ.tile <= (l1_cache*4)/16;
}
# L2 cache (for array A)
constraint mm_l2 {
    loopK.tile * loopI.tile <= (l2_cache*1024)/16;
}
# unroll constraint
constraint mm_unroll {
    (loopI.unroll * loopJ.unroll * loopK.unroll) <= register_file_size;
}
# putting everything together
specification {
    mm_l1 && mm_l2 && mm_unroll;
}
}
```

Table A.4: Python script output for MM specification

```
# import the python contraint module
from constraint import *
tiling_mm=Problem()

# Constant Declarations
# type:: int
l1_cache=128

# type:: int
l2_cache=4096

# type:: int
register_file_size=16

# parameter Declarations
class tile:
default=32
def values(self):
    ls=[]
    for i in range(1,8):
        ls.append(pow(2,i))
return ls

# Region Set association: loop
tiling_mm.addVariable("loopI_tile", tile().values())
tiling_mm.addVariable("loopJ_tile", tile().values())
tiling_mm.addVariable("loopK_tile", tile().values())

class unroll:
    default=1
    def values(self):
        ls=[]
        ls=range(1, 8, 2)
        return ls

# Region Set association: loop
tiling_mm.addVariable("loopI_unroll", unroll().values())
tiling_mm.addVariable("loopJ_unroll", unroll().values())
tiling_mm.addVariable("loopK_unroll", unroll().values())

# Constraint Declarations
def mm_l1 (loopJ_tile,loopK_tile):
    return ((loopK_tile * loopJ_tile) <= ((l1_cache * 1024) / 16))

def mm_l2 (loopK_tile,loopI_tile):
    return ((loopK_tile * loopI_tile) <= ((l2_cache * 1024) / 16))

def mm_unroll (loopK_unroll,loopJ_unroll,loopI_unroll):
    return (((loopI_unroll * loopJ_unroll) * loopK_unroll) <= register_file_size)

# Specification
def specification(loopK_unroll,loopJ_unroll,loopI_unroll,loopJ_tile,
                  loopK_tile,loopI_tile):
    return ((mm_l1(loopJ_tile, loopK_tile)  and
        mm_l2(loopK_tile, loopI_tile)) and
        mm_unroll(loopK_unroll, loopJ_unroll, loopI_unroll))

tiling_mm.addConstraint(FunctionConstraint(specification), \
("loopK_unroll","loopJ_unroll","loopI_unroll","loopJ_tile","loopK_tile","loopI_tile"))

# format the output and print solutions
solution = tiling_mm.getSolution()
```

Bibliography

[1] D. Abramson, A. Lewis, T. Peachey, and C. Fletcher. An automatic design optimization tool and its application to computational fluid dynamics. In *Proceedings of Supercomputing '01*, pages 25–25, November 2001.

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: tools for performance analysis of optimized parallel programs. *Concurr. Comput. : Pract. Exper.*, 22(6):685–701, 2010.

[3] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[4] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[5] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM*, 45(6):891–923, 1998.

[6] S. Baden. Kelp distribution webpage. `http://cseweb.ucsd.edu/groups/hpcl/scg/KeLP1.4/`. [last accessed: Feb, 2010].

[7] D. Bedard, M. Y. Lim, R. Fowler, and A. Porterfield. Powermon: Fine-grained and integrated power monitoring for commodity computer systems. In *Proceedings of the IEEE SoutheastCon '10*, pages 479 –484, 2010.

[8] F. Berman and R. Wolski. Scheduling From the Perspective of the Application. In *Proceedings of the 5th International ACM Symposium on High Performance Parallel and Distributed Computing*, page 100, 1996.

[9] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Program Optimization System. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2008.

[10] P. N. Brown, R. D. Falgout, and J. E. Jones. Semicoarsening multigrid on distributed memory machines. *SIAM Journal on Scientific Computing*, 21(5):1823–1834, 2000.

[11] C. Cascaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski. Multiple page size modeling and optimization. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 339–349, 2005.

[12] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, and R. Vuduc. Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures. In *Proceedings of the 25th International Parallel and Distributed Processing Symposium*, pages 1 –12, 2010.

[13] Y. Che, Z. Wang, and X. Li. Reduction Transformations for Optimization Parameter Selection. In *Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region*, page 281, 2005.

[14] C. Chen. *Model-Guided Empirical Optimization for Memory Hierarchy.* PhD thesis, University of Southern California, 2007.

[15] C. Chen, J. Chame, and M. Hall. CHiLL: A Framework for Composing High-level Loop Transformations. Technical report, University of Southern California, 2008.

[16] C. Chen, J. Chame, and M. W. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2005.

[17] J. Choi and J. J. Dongarra. Scalable linear algebra software libraries for distributed memory concurrent computers. In *Proceedings of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems*, page 170, 1995.

[18] I. Chung and J. K. Hollingsworth. Using Information from Prior Runs to Improve Automated Tuning Systems. In *Proceedings of Supercomputing '04*, 2004.

[19] I. Chung and J. K. Hollingsworth. A Case Study Using Automatic Performance Tuning for Large-Scale Scientific Programs. In *Proceedings of the 15th International ACM Symposium on High Performance Parallel and Distributed Computing*, pages 45–56, 2006.

[20] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, 23(1):7–22, 2002.

[21] M. E. Crovella and A. Bestavros. Self-similarity in world wide web traffic: evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5:835–846, December 1997.

[22] M. E. Crovella and M. S. Taqqu. Estimating the heavy tail index from scaling properties. *Method. Comput. Appl. Prob.*, 1(1):55–79, 1999.

[23] C. Ţăpuş, I. Chung, and J. K. Hollingsworth. Active harmony: towards automated performance tuning. In *Proceedings of Supercomputing '02*, pages 1–11, November 2002.

[24] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of Supercomputing '08*, pages 4:1–4:12, November 2008.

[25] J. E. Dennis, Jr. and V. Torczon. Direct Search Methods on Parallel Machines. *SIAM Journal of Optimization*, 1(4):448–474, 1991.

[26] W. Dorland, F. Jenko, M. Kotschenreuther, and B. Rogers. Electron temperature gradient turbulence. *Physics Review Letters*, 85(26):5579–82, 2000.

[27] J. B. Drake, S. Hammond, R. James, and P. H. Worley. Performance tuning and evaluation of a parallel community climate model. In *Proceedings of Supercomputing '99*, page 34, November 1999.

[28] J. K. Dukowicz, R. D. Smith, and R. C. Malone. A reformulation and implementation of the bryan-cox-semtner ocean model. *Journal of Atmospheric and Oceanic Technology*, 10:195–208, 1993.

[29] X. Feng, R. Ge, and K. Cameron. Power and energy profiling of scientific applications on distributed systems. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium*, page 34, 2005.

[30] M. Frigo. A fast Fourier transform compiler. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1999.

[31] G. Fursin and A. Cohen. Building a practical iterative compiler. In *Workshop on Statistical and Machine Learning Approaches to Architectures and Compilation (SMART'09)*, 2007.

[32] G. A. Geist II, J. A. Kohl, and P. M. Papadopoulos. CUMULVS: Providing fault tolerance, visualization, and steering of parallel applications. *International Journal of High Performance Computing Applications*, 11:224–236, 1997.

[33] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, 2006.

[34] Goto-Webpage. `http://www.tacc.utexas.edu/resources/software`. [last accessed: October 5, 2007].

[35] A. Hartono and S. Ponnuswamy. Annotation-Based Empirical Performance Tuning Using Orio. In *Proceedings of the 23rd International Parallel and Distributed Processing Symposium*, May 2009.

[36] J. K. Hollingsworth and P. J. Keleher. Prediction and adaptation in active harmony. *Cluster Computing*, 2:195–205, July 1999.

[37] R. Hooke and T. A. Jeeves. "Direct Search" Solution of Numerical and Statistical Problems. *Journal of the ACM*, 8(2):212–229, 1961.

[38] S. Hunold and T. Rauber. Automatic tuning of PDGEMM towards optimal performance . In *Proceedings of European Conference on Parallel Computing*, pages 837–846, August 2005.

[39] E. Ipek, B. R. de Supinski, M. Schulz, and S. A. McKee. An Approach to Performance Prediction for Parallel Applications. In *Proceedings of European Conference on Parallel Computing*, August 2005.

[40] J. Joines and C. Houck. On the use of non-stationary penalty functions to solve nonlinear constrained optimization problems with GA's. In *IEEE Conference on Evolutionary Computation*, volume 2, pages 579–584, June 1994.

[41] P. W. Jones, P. H. Worley, Y. Yoshida, I. J. B. White, and J. Levesque. Practical performance portability in the parallel ocean program (pop): Research articles. *Concurrency and Computation: Practice and Experience*, 17(10):1317–1327, 2005.

[42] J. Jorba, T. Margalef, and E. Luque. Search of performance inefficiencies in message passing applications with KappaPI 2 tool. In *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699, pages 409–419. Springer Berlin / Heidelberg, 2007.

[43] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library interface guide. Technical Report CS-TR-3445, University of Maryland at College Park, 1995.

[44] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of Supercomputing '01*, November 2001.

[45] D. Kim, L. Renganarayanan, D. Rostron, S. Rajopadhye, and M. M. Strout. Multi-level tiling: M for the price of one. In *Proceedings of Supercomputing '07*, pages 1–12, November 2007.

[46] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, page 237, 2000.

[47] T. G. Kolda, R. M. Lewis, and V. Torczon. Optimization by Direct Search: New Perspectives on Some Classical and Modern Methods. *SIAM Review*, 45(3):385–482, 2004.

[48] M. Kotschenreuther, G. Rewoldt, and W. M. Tang. Comparison of initial value and eigenvalue codes for kinetic toroidal plasma instabilities. *Computer Physics Communications*, 88:128–140, August 1995.

[49] W. T. C. Kramer and C. Ryan. Performance Variability of Highly Parallel Architectures. In *International Conference on Computational Science*, pages 560–569, 2003.

[50] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, volume 38, pages 12–23, New York, NY, USA, 2003. ACM.

[51] J. C. Lagarias, J. A. Reeds, M. H. Wright, and P. E. Wright. Convergence properties of the Nelder-Mead simplex algorithm in low dimensions. *SIAM Journal on Optimization*, 9:112–147, 1998.

[52] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 249–258, New York, NY, USA, 2007. ACM.

[53] R. M. Lewis and V. Torczon. Rank Ordering and Positive Bases in Pattern Search Algorithms. Technical Report TR-96-71, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA, 1996.

[54] C. Liao, D. J. Quinlan, R. Vuduc, and T. Panas. Effective Source-to-Source Outlining to Support Whole Program Empirical Optimization. In *Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing*, Newark, Delaware, 2009.

[55] D. Martin and K. Cartwright. Solving poisson's equation using adaptive mesh refinement. Technical Report UCB/ERL M96/66, EECS Department, University of California, Berkeley, 1996.

[56] K. I. M. McKinnon. Convergence of the Nelder–Mead Simplex Method to a Nonstationary Point. *SIAM Journal on Optimization*, 9(1):148–158, 1998.

[57] A. Morajko, P. Caymes-Scutari, T. Margalef, and E. Luque. MATE: Monitoring, Analysis and Tuning Environment for parallel distributed applications. *Concurrency and Computation: Practice and Experience*, 19(11):1517–1531, 2007.

[58] D. M. Mount. Ann webpage. `http://www.cs.umd.edu/~mount/ANN/`. [last accessed: Feb 09, 2009].

[59] R. Mraz. Reducing the variance of point to point transfers in the ibm 9076 parallel computer. In *Proceedings of the 1994 conference on Supercomputing*, Supercomputing '94, pages 620–629, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[60] I. Neamtiu. *Practical Dynamic Software Updating.* PhD thesis, University of Maryland, College Park, August 2008.

[61] J. Nelder and R. Mead. A Simplex Method for Function Minimization. *Computer Journal*, 7:308–313, 1965.

[62] Y. L. Nelson, B. Bansal, M. Hall, A. Nakano, , and K. Lerman. Model-guided performance tuning of parameter values: A case study with molecular dynamics visualization. In *Proceedings of the 22nd International Parallel and Distributed Processing Symposium*, pages 1–8, April 2008.

[63] Nersc. National energy research scientific computing center. `www.nersc.gov`.

[64] G. Niemeyer. python-constraint module. `http://labix.org/python-constraint.org`.

[65] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. *ACM SIGOPS Operating Systems Review*, 31(5):276–287, 1997.

[66] F. Otto, C. A. Schaefer, M. Dempe, and W. F. Tichy. A language-based tuning mechanism for task and pipeline parallelism. In *Proceedings of European Conference on Parallel Computing*, Euro-Par'10, pages 328–340, Berlin, Heidelberg, 2010. Springer-Verlag.

[67] S. G. Parker and C. R. Johnson. SCIRun: A Scientific Programming Environment for Computational Steering. In *Proceedings of Supercomputing '95*, November 1995.

[68] T. Parr. Antlrv3. `http://www.antlr.org`.

[69] F. Petrini, D. J. Kerbyson, and S. Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Proceedings of Supercomputing '03*, November 2003.

[70] POP-Webpage. Pop benchmark inputs. `http://oceans11.lanl.gov/COSIMdownloads/POPBenchmarkInputs/`. [last accessed: Oct 06, 2007].

[71] L. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time . In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2008.

[72] A. Qasem. *Automatic Tuning of Scientific Applications.* PhD thesis, Rice University, 2007.

[73] A. Qasem and K. Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *Proceedings of the 2006 ACM International Conference on Supercomputing*, pages 249–258, New York, NY, USA, 2006. ACM.

[74] A. Qasem, K. Kennedy, and J. Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. *The Journal of Supercomputing*, 36(2):183–196, 2006.

[75] R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed. Autopilot: Adaptive Control of Distributed Applications. In *Proceedings of the 7th International ACM Symposium on High Performance Parallel and Distributed Computing*, page 172, 1998.

[76] G. Rivera and C. Tseng. Tiling optimizations for 3D scientific computations. In *Proceedings of Supercomputing '00*, 2000.

[77] S. Ros. "state of the art expression evaluation". `http://www.codeproject.com/KB/recipes/sota_expression_evaluator.aspx`, November 2007.

[78] S. S. Shende and A. D. Malony. The TAU Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.

[79] K. Singh, E. İpek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana. Predicting parallel application performance via machine learning approaches: Research articles. *Concurrency and Computation: Practice and Experience*, 19(17):2219–2235, 2007.

[80] R. D. Smith, J. K. Dukowicz, and R. C. Malone. Parallel ocean general circulation modeling. In *Proceedings of the Eleventh Annual International Conference of the Center for Nonlinear Studies on Experimental mathematics : Computational Issues in Nonlinear Science*, pages 38–61. Elsevier, 1992.

[81] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A Framework for Application Performance Modeling and Prediction. In *Proceedings of Supercomputing '02*, November 2002.

[82] Swig. Simplified wrapper and interface generator. `http://www.swig.org`.

[83] V. Tabatabaee, A. Tiwari, and J. K. Hollingsworth. Parallel Parameter Tuning for Applications with Performance Variability. In *Proceedings of Supercomputing '05*, page 57, November 2005.

[84] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 204–215, 2003.

[85] H.-L. Truong and T. Fahringer. Scalea: A performance analysis tool for distributed and parallel programs. In *Euro-Par 2002 Parallel Processing*, volume 2400, pages 41–55. Springer Berlin / Heidelberg, 2002.

[86] J. S. Vetter and P. H. Worley. Asserting performance expectations. In *Proceedings of Supercomputing '02*, pages 1–13, November 2002.

[87] M. J. Voss and R. Eigenmann. ADAPT: Automated De-coupled Adaptive Program Transformation. In *Proceedings of the International Conference on Parallel Processing, ICCP '00*, pages 163–170, August 2000.

[88] R. Vuduc, J. W. Demmel, and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16:521–530, June 2005.

[89] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of Supercomputing '98*, 1998.

[90] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Lattice boltzmann simulation optimization on leading multicore platforms. In *Proceedings of the 22nd International Parallel and Distributed Processing Symposium*, pages 1 –14, 2008.

[91] P. H. Worley and I. T. Foster. Parallel spectral transform shallow water model: a runtime–tunable parallel benchmark code. In J. J. Dongarra and D. W. Walker, editors, *Proc. Scalable High Performance Computing Conf.*, pages 207–214. 1994.

[92] X. Wu, V. Taylor, C. Lively, and S. Sharkawi. Performance analysis and optimization of parallel scientific applications on cmp cluster systems. In *ICPPW '08: Proceedings of the 2008 International Conference on Parallel Processing - Workshops*, pages 188–195, 2008.

[93] J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A language and compiler for DSP algorithms. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2001.

[94] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. POET: Parameterized Optimizations for Empirical Tuning. *Proceedings of the 21st International Parallel and Distributed Processing Symposium*, pages 1–8, March 2007.

[95] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is Search Really Necessary to Generate High-Performance BLAS? *Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Platform Adaptation*, 93(2):358–386, 2005.

[96] D. Yu and S. S. Girimaji. Multi-block Lattice Boltzmann Method: Extension to 3D and Validation in Turbulence. *Physica A: Statistical Mechanics and its Applications*, 362(1):118 – 124, 2006.